

# **Simulation von Fehlern in digitalen Schaltungen mit SystemC**

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik  
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften  
(Dr.-Ing.)

genehmigte Dissertation

vorgelegt von

Dipl.-Ing.  
Silvio André Misera

geboren am 18. Dezember 1969 in Finsterwalde

Gutachter: Prof. Dr.-Ing. Heinrich Theodor Vierhaus

Gutachter: Prof. Dr.-Ing. Sorin A. Huss

Gutachter: Prof. Dr.-Ing. habil. Bernd Straube

Tag der mündlichen Prüfung: 05. Dezember 2007



# Danksagung

Diese Dissertation entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl Technische Informatik der Brandenburgischen Technischen Universität Cottbus. Mein ganz besonderer Dank gebührt meinem Doktorvater Prof. Heinrich Theodor Vierhaus für die wissenschaftliche Betreuung, die eingeräumten Freiräume und die konstruktive Kritik.

Ebenso möchte ich mich bei den Kollegen am Lehrstuhl für die angenehme Arbeitsatmosphäre bedanken, insbesondere bei unserer Sekretärin Kathleen Nörenberg für ihre nützlichen Hinweise und organisatorischen Hilfestellungen und bei Rene Kothe, welcher immer Zeit und Geduld für fachliche Gespräche und Anregungen fand.

Mein spezieller Dank gilt den Studenten André Sieber, Roberto Urban, Stephan Schulz, Lars Breitenfeld, Mario Pink, Elnar Hajiev und Michael Waldmann, die durch ihre Tätigkeit als studentische Hilfskraft oder durch Studienarbeiten Beiträge und Anregungen zu dieser Arbeit lieferten.

Besonderer Dank gebührt Kay Buchelt für das geduldige und schnelle Korrekturlesen.

Abschließend bedanke ich mich beim Promotionsausschuss, bestehend aus den Gutachtern Prof. Bernd Straube von der Fraunhofer Gesellschaft Dresden, Prof. Sorin A. Huss von der Technischen Universität Darmstadt und natürlich meinem Doktorvater. Außerdem bedanke ich mich bei den weiteren Mitgliedern Prof. Peter Bachmann, Prof. Jörg Nolte und besonders beim Vorsitzenden Prof. Hartmut König für die Koordination des Promotionsverfahrens.



# Inhaltsverzeichnis

<b>Inhaltsverzeichnis .....</b>	<b>V</b>
<b>1 Einleitung .....</b>	<b>1</b>
1.1 Motivation und Zielsetzung .....	1
1.2 Struktur der Arbeit .....	5
1.3 Publikationen .....	7
<b>2 Systementwurf und SystemC .....</b>	<b>9</b>
2.1 Entwurfsebenen .....	9
2.2 Einführung zu SystemC .....	12
2.3 SystemC im Systementwurf .....	13
2.4 Modellierung mit SystemC .....	17
2.5 SystemC-Projektumgebung .....	19
<b>3 Simulation .....</b>	<b>21</b>
3.1 Logiksimulation .....	22
3.1.1 Kompilierte Simulation .....	22
3.1.2 Interpretierende Simulation .....	23
3.1.3 Native-code-compiled Simulation .....	23
3.1.4 Tabellengesteuerte Simulation .....	24
3.1.5 Simulationstypen .....	24
3.1.6 SystemC-Simulationskernel .....	26
3.2 Fehlersimulation .....	27
3.2.1 Serielle Fehlersimulation .....	30
3.2.2 Parallele Fehlersimulation .....	31
3.2.3 Deduktive Fehlersimulation .....	33
3.2.4 Concurrent Fehlersimulation .....	34
3.3 Bisherige Fehlersimulationen in SystemC .....	35
<b>4 Hardware- und Fehlermodellierung .....</b>	<b>37</b>
4.1 Hardware-Modellierung in SystemC .....	39
4.1.1 Die Register-Transfer-Ebene in SystemC .....	39
4.1.2 Die Logik-Ebene in SystemC .....	41
4.1.3 Die Schalterebene in SystemC .....	42
4.2 Ausgewählte Hardware-Komponenten .....	47

4.2.1 CMOS-Komplexgatter .....	47
4.2.2 Pass-Transistor-Logik.....	51
4.2.3 Transfer-Gatter .....	53
4.2.4 Simulationsexperimente auf der Schalterebene.....	56
4.3 Fehler und deren Modellierung in SystemC.....	57
4.3.1 Fehlertypen und Fehlerursachen.....	58
4.3.2 Fehlermodelle .....	60
<b>5 Fehlerinjektion in digitale Hardwaremodelle.....</b>	<b>77</b>
5.1 Verlässlichkeit und Fehlereffekte .....	78
5.1.1 Verlässlichkeit .....	78
5.1.2 Das Auftreten von Fehlern .....	79
5.1.3 Der Fehlerraum.....	80
5.1.4 Das FARM-Set .....	81
5.2 Fehlerinjektionen .....	82
5.2.1 Gruppen der Fehlerinjektion.....	82
5.3 Simulierte Fehlerinjektion in HDL-Beschreibungen.....	85
5.3.1 Bisherige Fehlerinjektionen mit SystemC.....	87
5.4 Fehlerinjektionstechniken in SystemC-Beschreibungen .....	95
5.4.1 Die Saboteur-Technik.....	95
5.4.2 Die Mutant-Technik .....	103
5.4.3 Simulationskommandos .....	114
5.4.4 Simulationsvergleich der Fehlerinjektions-Techniken.....	124
<b>6 Mixed-Language-Co-Simulation.....</b>	<b>125</b>
6.1 Das Fault Injection Tool .....	128
6.2 Konzept der Mixed-Language-Co-Simulation .....	131
6.3 Modifikationen an der SystemC-Bibliothek.....	131
6.4 Die Kopplung in einem Simulationsprojekt .....	134
6.5 Simulationsergebnisse .....	138
<b>7 Simulationsbeschleunigung durch Parallelität .....</b>	<b>141</b>
7.1 Wichtige Kenngrößen des parallelen/verteilten Rechnens.....	146
7.2 Beschleunigung durch paralleles Rechnen auf einem Einzelprozessorsystem ..	149
7.2.1 Bit-parallele Simulation.....	149
7.2.2 Simulationen mit vierwertiger Logik .....	153
7.2.3 Parallelisierung auf höherer Ebene.....	155
7.3 Beschleunigung durch verteiltes Rechnen.....	159
7.3.1 Message Passing Interface (MPI).....	160
7.3.2 Aufbau des Rechnersystems.....	161

7.3.3 Multi-Processing-Simulation (MPS) .....	162
7.3.4 Ablauf der verteilten Simulation .....	165
7.3.5 Beispielsimulationen .....	167
7.4 Fazit des beschleunigten Rechnens .....	170
<b>8 Zusammenfassung und Ausblick .....</b>	<b>173</b>
<b>Anhang.....</b>	<b>177</b>
A.1 Modelle und Beispiele .....	177
A.2 Code-Mutationen in verhaltensorientierten HDL-Modellen .....	194
A.3 Überblick zu den Simulationsmodellen.....	198
A.4 MPS-Klassendiagramm .....	199
A.5 grafische SystemC-Darstellungen .....	200
<b>Abkürzungsverzeichnis .....</b>	<b>201</b>
<b>Symbolverzeichnis .....</b>	<b>203</b>
<b>Abbildungsverzeichnis .....</b>	<b>205</b>
<b>Tabellenverzeichnis .....</b>	<b>207</b>
<b>Listings.....</b>	<b>209</b>
<b>Literaturverzeichnis .....</b>	<b>211</b>





# 1 Einleitung

*„Siehe, es ist einerlei Volk und einerlei Sprache unter ihnen allen  
und dies ist der Anfang ihres Tuns;  
nun wird ihnen nichts mehr verwehrt werden können von allem,  
was sie sich vorgenommen haben zu tun.“  
1.Mose 11.6 (Turmbau zu Babel)*

## 1.1 Motivation und Zielsetzung

Elektronische Systeme sind im heutigen Alltag allgegenwärtig. Dabei hat deren Entwicklung und Verbreitung seit der Realisierung der ersten integrierten Schaltung im Jahre 1959 ein atemberaubendes Tempo eingenommen. Insbesondere Mikroprozessoren stellen in solchen Systemen den Kern der Applikation dar. Zum einen ist deren Verwendung in Personal Computern oder in modernen Mobiltelefonen offensichtlich. Zum anderen steckt aber eine große Anzahl an Prozessoren in Systemen verborgen, wo der Anwender diese nicht gleich vermutet. Mittlerweile besitzt die Elektronik in einem Automobil einen deutlichen Anteil an der Komponentenzahl und man schätzt, dass deren Beitrag an den Herstellungskosten in Zukunft bis zu 40 % betragen wird. Solche Systeme werden als Eingebettete Systeme (engl. *embedded systems*) bezeichnet und sind dadurch gekennzeichnet, dass sie ihre Aufgaben unauffällig ausüben. Aber nicht nur im Automobil, sondern auch in der Waschmaschine oder an der Ampel verrichten sie ihren Dienst. Mittlerweile gibt es sogar Bekleidung und Laufschuhe mit Mikroprozessor-gesteuerter Elektronik.

Für die Entwickler von elektronischen Systemen ergibt sich dabei eine Reihe von Herausforderungen. So soll ein neues System meist einen geringeren Platzbedarf, einen geringeren Stromverbrauch und eine höhere Geschwindigkeit bei gleichem oder geringerem Preis als das Vorgängermodell aufweisen. Der viel zitierte Gordon E. Moore, ein Mitbegründer der Intel Corporation, hatte 1965 vorausgesagt, dass sich ca. alle zwei Jahre die Anzahl der Schaltelemente auf einem Chip verdoppeln wird und somit der diesbezügliche Fortschritt ein exponentielles Wachstum besitzt. Später modifizierte er diese Aussage sogar noch dahin gehend, dass sich alle 18 Monate die Transistoranzahl auf einem Schaltkreis verdoppelt. Dieses als *Moore's Law* bekannte Gesetz hat, trotz früherer gegensätzlicher Schätzungen, bis heute seine Gültigkeit

behalten und es wird auch noch für die nächsten Jahre gültig sein. Das Streben nach der Einhaltung von Moore's Law besitzt aber noch einen weiteren Nebeneffekt, so verdoppelt sich beispielsweise alle vier Jahre der Preis für eine neue Chipfabrik.

Der Fortschritt bzw. die Herausforderung liegt jedoch nicht nur in der eher öffentlich erwähnten möglichen Fertigung immer komplexerer Systeme, sondern auch in deren Entwurf. Der komplexe Entwurf hat diverse Facetten. Wurden in der Vergangenheit verschiedene Schaltkreise mit einer bestimmten Aufgabe auf einer Leiterplatte (PCB) vereint, so sind heute die unterschiedlichsten Funktionen auf einem einzigen Stück Silizium untergebracht, den sogenannten *Systems on a Chip* (SoCs) oder *Systems in Package* (SIP). Ein SoC besitzt eine recht inhomogene Struktur. Neben einem Prozessorkern sind Speicher, weitere Signalprozessoren, spezielle HW-Komponenten für schnelle Ausführungen und u. U. noch analoge Baugruppen vorhanden. Diese kommunizieren miteinander über ein kompliziertes Netzwerk. Aber der komplexe Entwurf ist auch durch die Zusammenarbeit vieler Ingenieure gekennzeichnet, die in Teams für die unterschiedlichen Aufgabenbereiche (z. B. Hardware oder Software) organisiert sind. Ein weiterer Punkt ist für den Entwurf von wesentlicher Bedeutung: Der Zeitraum von der ersten Idee bis zur Markteinführung eines Produkts, häufig als *time to market* bezeichnet, trägt entscheidend zum Erfolg bei.

Um dieses Komplexitätsproblem in den Griff zu bekommen, darf die Entwicklung der Entwurfswerkzeuge nicht vernachlässigt werden. Traditionell war der Designprozess in den 90er Jahren so geprägt, dass man in den ersten Entwurfsebenen auf Programmiersprachen wie C++ zurückgriff. Mit der weiteren Verfeinerung und der Aufteilung der Aufgaben auf HW und SW war spätestens auch der Einsatz von Werkzeugen der Hardware-Modellierung notwendig. Hier entstand eine wesentliche Lücke in den Modellen und Abstraktionen zwischen der HW- und der SW-Seite, die manuelle „Transfers“ erforderte. Am Ende der 90er Jahre entstand eine Initiative, welche versuchte, die fehlende Möglichkeit der HW-Modellierung in C++ zu implementieren. Das Ergebnis der Bemühungen kennen wir heute als SystemC, welches 1999 von der Open SystemC Initiative (OSCI) aus der Taufe gehoben wurde. Mittlerweile bekam es in der IEEE 1666 seine Standardisierung. SystemC ist dabei eine Erweiterung von C++ in Form einer Klassenbibliothek, welche eine ganze Reihe an Mitteln bietet, um Hardware zu modellieren. Die Beschreibung eines komplexen Systems mit den Mitteln einer mächtigen Programmiersprache ist jedoch nur ein Fakt. Ursprünglich waren und sind Programmiersprachen dazu gedacht, ausführbare Programme zu erzeugen. Mit SystemC hat man also nicht nur die Gelegenheit ein System zu beschreiben, sondern es auch zu kompilieren, auszuführen und so zu simulieren. Da SystemC auf unterschiedlichen Entwurfsebenen anzutreffen ist - man verfolgte bei der SystemC-Idee von vornherein das Ziel eines durchgehenden Designflows - können diverse Teams auf ungleichen Stufen und in unterschiedlichen Domänen gleichzeitig an einem Projekt arbeiten.

Mit der Umsetzung einer Idee in ein greifbares Objekt ist der komplexe Systementwurf aber noch nicht getan. Es besteht auch die Forderung, dass das System

die ihm zugewiesenen Aufgaben zuverlässig erfüllt. Kein Produktionsprozess kann eine 100-prozentige Garantie der Korrektheit geben. Gelangen fehlerhafte Bauelemente in den Handel und später zum Einsatz in einem größeren System, so wird die Fehlersuche und -behebung deutlich teurer, als wenn man dieses Teil schon vorher ausgesondert hätte. Außerdem trägt die Auslieferung fehlerhafter Komponenten nicht zum guten Ruf eines Unternehmens bei. Um solche Probleme zu vermeiden, werden elektronische Bauteile während und nach der Fertigung getestet. Je früher die Aussonderung erfolgt - umso besser. Der Test findet üblicherweise durch einen Betrieb mit bestimmten Eingabemustern statt und diese werden meist durch Werkzeuge der automatischen Testmustererzeugung (ATPG) erstellt. Ein wichtiges Mittel zur Validierung der Muster stellt der Fehlersimulator dar. So dient sie beispielsweise zum Aufspüren weiterer Fehlerfälle für ein gegebenes Eingabemuster. Die klassische Fehlersimulation unterscheidet sich von anderen Simulationen (z. B. Logiksimulationen) dadurch, dass sie für eine große Anzahl von möglichen Schaltungsfehlern in möglichst kurzer Zeit Ergebnisse liefert.

Mit der Auslieferung und der Verwendung fehlerfreier Bauteile ist das Problem noch nicht aus der Welt. Während des Betriebes können immer noch Fehler auftreten, z. B. bedingt durch äußere Störungen oder Alterungsprozesse. In einem solchen Fall ist das Versagen eines Telefons oder eines Unterhaltungsgerätes störend, aber meistens noch ungefährlich. Es existieren aber zahlreiche Anwendungen, insbesondere für embedded systems, wo der Ausfall kritisch ist. Solche sicherheitskritischen Systeme befinden sich mittlerweile in fast allen maschinenbetriebenen Fortbewegungsmitteln, aber auch in Kraftwerken oder in Verteidigungssystemen. Damit diese Systeme auch beim Auftreten eines Fehlers nicht versagen, besitzen sie zusätzliche Mechanismen der Fehlertoleranz. Für den Entwurf sicherheitskritischer Anwendungen besteht die Anforderung, diese Fehlertoleranz auch nachzuweisen. In diesem Bereich hat sich die Fehlerinjektion etabliert.

Die Simulation von Fehlern in elektronischen Systemen besitzt somit zwei Kernaufgaben. Zum einen dient sie der Validierung von Testmustern für den Fertigungsprozess. Dieses Feld wird üblicherweise mit dem Begriff Fehlersimulation bezeichnet. Zum anderen dient die Simulation von Fehlern der Untersuchung des Verhaltens eines Systems beim Auftreten eines Fehlers im Betrieb. Dieser Bereich ist gemeinhin als Fehlerinjektion bzw. genauer als simulierte Fehlerinjektion bekannt.

Während für die Fehlerinjektion auch normale Logik- bzw. HDL-Simulatoren benutzt werden, finden bei der Fehlersimulation häufig besondere Werkzeuge ihren Einsatz, die häufig hochspezielle Algorithmen zur schnellen Aufgabenlösung verwenden. Typisch ist für diese Simulatoren, dass sie auf ein spezielles Gebiet zugeschnitten sind, z. B. für die Logikebene und nur einen begrenzten Umfang an simulierbaren Komponenten bzw. Modellen haben.

Zur Modellierung und Simulation komplexer Systeme hat SystemC seit ca. dem Jahr 2000 zunehmend an Bedeutung gewonnen. Dies beruht nicht nur darauf, dass es nicht kommerziell ist, kostenlos zur Verfügung steht und auf dem Know-how vieler Fachleute beruht. Es bietet auch den vollen Sprachumfang von C++ und erlaubt einen durchgehenden Entwurfsablauf mit Simulationen bis mindestens zur Register-Transfer-Ebene ohne aufwändige Sprachkonvertierungen. Für die Entwurfsvalidierung gibt es auch den Bedarf an Simulatoren, welche Schaltungen mit eingebauten Fehlern berechnen. Damit soll die effiziente Validierung des Verhaltens eines komplexen Systems für relevante Fehlerfälle möglich sein. Lösungen, die zum einen auf SystemC basieren und zum anderen Schaltungen mit Fehlern simulieren, gibt es bisher kaum. Andererseits sind für andere Hardwarebeschreibungssprachen wie VHDL eine ganze Reihe an Arbeiten und systematischen Untersuchungen zur Fehlerinjektion bekannt. Es existieren sogar einige Fehlersimulationsanwendungen auf der Basis von HDL-Simulatoren. Für SystemC existieren hierzu nur wenige Veröffentlichungen und diese sind auf ein spezielles Umfeld beschränkt.

Ausgehend von diesem gegebenen Defizit entstand die Idee, die beiden Anwendungsfelder des SystemC-Entwurfs und der Simulation von Fehlern miteinander zu verknüpfen. Ziel sollte es sein, verschiedene Möglichkeiten des Fehlereinbaus zu finden und diese systematisch bezüglich ihrer Effektivität und Effizienz näher zu untersuchen. Mit dem Einsatz anderer HDLs gab es bereits erfolgreiche Applikationen. Eine Simulation von Fehlern scheint somit auch mit SystemC möglich. Zum einen erlaubt SystemC eine oft deutlich schnellere Ausführung, da das Modell als ausführbares Programm vorliegt und zum anderen bietet SystemC auf der Basis von C++ eine große Mächtigkeit an Sprachmitteln, jenseits von Hardwarebeschreibungssprachen. Da heutige Systeme häufig HW- und SW-Anteile vereinen, sich dadurch aber der Betrachtungsumfang deutlich steigert, wurde in dieser Arbeit eine Einschränkung auf die Modellierung von digitaler Hardware vorgenommen.

Da SystemC außerdem die Möglichkeit offeriert, den Vorteil eines durchgehenden Entwurfsprozesses zu bieten, galt es auch für eine genauere Modellierung die Möglichkeiten unterhalb der Register-Transfer-Ebene näher zu untersuchen. Modellbetrachtungen unterhalb der Register-Transfer-Ebene sind insbesondere für Fehlerverhalten notwendig, um überhaupt eine ausreichende Sicherheit zum realitätsnahen Verhalten der Schaltung zu erhalten.

Wächst eine Schaltung linear, so reicht leider kein Rechensystem, welches ebenfalls nur über eine lineare Ausführungsbeschleunigung verfügt, um das Problem in gleicher Zeit zu lösen. Der Umfang der Fehlersimulation wächst z. B. mindestens quadratisch. Andererseits stehen aber nur die Computer zur Berechnung zur Verfügung, die schon entworfen und gebaut wurden. Um das Problem in machbarer Zeit zu lösen, sind andere Wege zur effizienten Lösungsfindung nötig. Auch wenn in SystemC eine Schaltungspartitionierung möglich ist und eine getrennte oder hierarchische Simulation hilft die Komplexität zu beherrschen, so ist ein weiteres Ziel die Untersuchung von Beschleunigungsmöglichkeiten für die Simulation von Fehlern in digitalen Schaltungen.

## 1.2 Struktur der Arbeit

Die vorliegende Arbeit ist in acht Kapiteln gegliedert. Das *erste Kapitel* umreißt den Hintergrund und die Motivation zu Schaltungssimulationen mit Fehlerinjektionen in SystemC-Modellen. Außerdem ist ein kurzer Überblick über eigene Publikationen wiedergegeben, die im Zusammenhang mit den Ergebnissen dieser Arbeit stehen.

Im *Kapitel 2* erfolgt zunächst eine einführende Beschreibung der Abstraktionsebenen des Systementwurfs. Außerdem sind die wichtigsten Merkmale von SystemC und zur Historie dieser Spracherweiterung aufgeführt. Eine Einführung zum Einsatz von SystemC in den verschiedenen Ebenen des Systementwurfs trägt zum Verständnis der später folgenden Ausführungen bei.

*Kapitel 3* zeigt anfangs eine Übersicht zur Simulation digitaler Schaltungen. In diesem Zusammenhang erfolgt auch eine Erklärung des SystemC-Simulationskernels. Betrachtungen zu den Problemen der Fehlersimulation und den verschiedenen Verfahren ergänzen dieses Kapitel.

Im *Kapitel 4* wird auf die Hardware- und Fehlermodellierung mit SystemC eingegangen. Es werden beispielhaft die Register-Transfer- und Gatterebene vorgestellt. Außerdem kommt es zu einer ausführlichen Vorstellung einer neuen SystemC-Modellierung auf der Schalterebene. Spezielle Hardware-Komponenten mit ihren besonderen Problemstellungen und deren Lösungen in SystemC sind ebenso Bestandteil des Kapitels.

Nach einer Einführung zum Wesen von Fehlererscheinungen und deren Klassifizierung erfolgen Darstellungen zu diversen gebräuchlichen Fehlermodellen. Eigene Lösungen zur Modellierung von Fehlern sind ebenfalls Inhalt dieses Kapitels.

Das *Kapitel 5* beschäftigt sich mit dem Kontext und der Notwendigkeit der Fehlerinjektion. Zum besseren Verständnis ist zunächst ein Überblick zum Stand der Technik wiedergegeben. Nennenswerte Implementierungen in SystemC werden ebenso vorgestellt. Anschließend kommt es zu einer ausführlichen Darstellung eigener Umsetzungen verschiedener Fehlerinjektions-Techniken in SystemC. Neben Techniken, die in ähnlicher Weise schon aus anderen HDLs bekannt sind, werden auch neue Lösungen vorgestellt und bewertet.

Im *Kapitel 6* wird eine Lösung präsentiert, mit der sich eine SystemC-Simulation mit anderen Applikationen koppeln lässt. Problemfelder solcher Kopplungen sind der zusätzliche Kommunikationsaufwand und die Synchronisation, welche deutlich messbar die Ausführungsgeschwindigkeit verschlechtern. Im Gegensatz dazu zeigt das präsentierte Beispiel an einem Mixed-Language-Projekt, dass es auch effiziente Lösungen gibt.

Das *Kapitel 7* beschäftigt sich mit verschiedenen Implementierungen, die eine Beschleunigung der Simulation von SystemC-Modellen zum Gegenstand haben. Dies wird durch verschiedene Formen der Parallelisierung erreicht. Neben den schon bekannten Verfahren aus dem Bereich der Fehlersimulation mit anderen Beschreibungsmitteln werden auch neue Methoden präsentiert, welche sich spezielle Eigenschaften von SystemC bzw. C++ zunutze machen. Außerdem sind diverse Implementierungen auf unterschiedlichen Ebenen der Parallelisierung dargestellt.

Eine Zusammenfassung zu den präsentierten Untersuchungen, Implementierungen und Bewertungen anhand von Ergebnissen ist im *Kapitel 8* wiedergegeben. Ein Ausblick bildet den Schlussteil.

Der *Anhang* beinhaltet weitere Informationen, wie z. B. Listings von diversen SystemC-Modellen, zu Programmen und eine Übersicht zu den Eigenschaften der Schaltungen in den Beispielsimulationen.

Zum besseren Verständnis beim Lesen der Arbeit werden im Folgenden einige Hinweise aufgeführt. Begriffe, die im Abschnitt eine besondere Bedeutung haben, sind kursiv dargestellt. Schlüsselwörter, welche im Nachfolgenden näher erläutert werden, sind zusätzlich fett hervorgehoben. Um auf eine Erläuterung oder einen Vergleich zu verweisen, der in einem anderen Textteil erläutert wird, erfolgt ein Hinweis darauf mit Klammern in denen ein Pfeil und eine Kapitel- und Abschnittennummerierung stehen. Zum Beispiel bedeutet (→3.2) – siehe Kapitel 3, Abschnitt 2.

Da sich diese Arbeit sehr mit SystemC-Modellen und Modifikationen der SystemC-Bibliothek beschäftigt, liegt es nahe, Programmcode in entsprechender C++/SystemC-Syntax zu verwenden. Dort wo es aus Platz- und Übersichtsgründen unangemessen ist, umfangreichen Code aufzuführen, werden kürzerer, selbsterklärender Pseudocode oder Abbildungen gebraucht. Begriffe aus dem Englischsprachigen, die mittlerweile eingedeutscht sind, findet man großgeschrieben vor.

## 1.3 Publikationen

Im Verlauf dieser Arbeit kam es zu einer Reihe von Publikationen, welche Ideen, Umsetzungen und Experimente zum Inhalt hatten. Dieser Abschnitt gibt einen kurzen Überblick mit Erläuterungen zu den Inhalten einiger internationaler Veröffentlichungen.

In „A Mixed Language Fault Simulation of VHDL and SystemC” [223] wird die Kopplung des VHDL-Programms FIT, welches Fehlerinjektionen in strukturellen VHDL-Beschreibungen erlaubt, mit einer SystemC-Simulation beschrieben. Hierzu wurden einige Veränderungen und Erweiterungen an der SystemC-Bibliothek vorgenommen. Durch den Einsatz der POSIX-Thread-Bibliothek ist diese Form der gekoppelten Simulation sehr effizient. Ansätze zur Injektion von Fehlern in das SystemC-Modell ergänzen diese Applikation.

In „Fault Injection Techniques and their Accelerated Simulation in SystemC” [224] werden verschiedene Fehlerinjektionstechniken für SystemC-HW-Modelle vorgestellt. Neben schon bekannten Ansätzen, die aus anderen HDL-Experimenten bekannt sind, kommt es auch zur Präsentation einer neuen Technik. Simulationsergebnisse unterstreichen die Leistungsfähigkeit dieser neuen Methode. Techniken zur kostengünstigen Beschleunigung der Simulation auf einem Einprozessorrechner sind ebenfalls Gegenstand dieses Papers. Neuartig ist auch die genauere Möglichkeit der Modellierung auf der Transistorebene in SystemC. Hierzu wurden SystemC-Klassen erweitert und Basiselemente kreiert. Mit der Simulation von Schaltungen, die aus Transmission-Gates bestehen, wird diese Modellierung hinsichtlich ihrer Effektivität bestätigt.

In „FIT - A Parallel Hierarchical Fault Simulator” [222] kommt es zu Vorstellung einer Umgebung zur Schaltungssimulation, die in verteilten Systemen ausgeführt werden kann. Zusätzlich erlaubt diese Umgebung die Simulation von Fehlern, wobei als Basis ein VHDL-Fehlerinjektionswerkzeug dient. Mit Hilfe eines Add-ons wird aus der Einzelplatzanwendung ein Mehrrechnersystem. Erste Ergebnisse zeigen, dass die Simulation mit Fehlern gut skaliert und praktikabel für ein verteiltes Rechnen ist. Der Einsatz des Systems in anderen größeren Anwendungen [229,223] unterstreicht die Leistungsfähigkeit der Parallelisierung.

Neben den vorgestellten Publikationen gab es noch andere [225,226,227,228]. In diesen Veröffentlichungen wurden weitere Beispiele zur effektiven Fehlerinjektion, Modelle der RT- bzw. Gatterebene mit Switch-level-Verhalten und Untersuchungen zur effektiven Fehlermodellierung vorgestellt.





## 2 Systementwurf und SystemC

SystemC (→2.2) gewinnt beim Entwurf von Systemen, die aus Hardware und Software bestehen, zunehmend an Bedeutung. Um die Rolle von SystemC innerhalb des Entwurfprozesses besser zu verstehen, werden im folgenden Abschnitt einige Grundlagen und Eigenschaften des modernen Systementwurfs erläutert.

Die Systeme, die im Zusammenhang mit dieser Arbeit interessieren, sind technischer Natur. Sie bestehen im Wesentlichen aus Komponenten der Elektrotechnik und der Informatik. Um mit ihrer Umgebung interagieren zu können, besitzen sie Schnittstellen, wie zum Beispiel Sensoren, Aktoren und signalverarbeitende Komponenten. Der signalverarbeitende Teil besteht hauptsächlich aus elektronischen Elementen.

Nach [189] ist der heutige Entwurf elektronischer Systeme von folgenden Aspekten geprägt:

- Die Komplexität und die Integrationsdichte nehmen ständig zu. Stets kleiner werdende Strukturen erhöhen die Packungsdichten und ermöglichen damit immer mehr Produkte mit komplexerer Elektronik. Die Leistungsfähigkeit der Elektronik steigt bezüglich Platzbedarf, Taktrate und Leistungsbedarf.
- Konkurrenzdruck und Kundenanforderungen erfordern immer kürzere Entwicklungszeiten. Eine verzögerte Markteinführung kann den Erfolg eines Unternehmens deutlich schmälern.
- Nur durch eine vollständige, widerspruchsfreie und verständliche Dokumentation sind die Wiederverwendung von Entwurfsdaten und die Produktwartung gegeben.

### 2.1 Entwurfsebenen

Die Komplexität elektronischer Systeme erfordert heutzutage eine systematisches Vorgehen bei ihrer Entwicklung. Bis das fertige Produkt vorliegt, arbeitet man zunächst mit einer Reihe von Modellen. Hierbei vereinfacht ein Modell das reale Systemverhalten und bildet die relevanten Eigenschaften im Verhalten nach. Je nach Einfachheitsgrad des Modells befindet man sich auf verschiedenen Abstraktionsebenen. Ausgehend von einer Aufgabenstellung, welche konkretisiert zu einer Spezifikation führt, findet der Entwurfseinstieg in der Systemebene statt. Zweckmäßig wird man die Schaltungsfunktion partitionieren und einzelnen Modulen zuweisen. Mit dem Übergang

zu tieferen Ebenen erfolgt eine zunehmende Verfeinerung durch die Implementierung weiterer Details. Das endgültige Ergebnis sind die notwendigen Fertigungsdaten für das elektronische System. Dies können Leiterplatten-Layouts, Masken für die Chipfertigung oder Programmierdaten für FPGAs sein.

Beim Entwurf unterscheidet man die drei Darstellungsweisen (Domains):

- Verhalten,
- Struktur und
- Geometrie.

Diese Darstellungs- bzw. Sichtweisen stehen über die Abstraktionsebenen (Levels) in Beziehung zueinander. Das Y-Diagramm in Abbildung 2.1 verdeutlicht dies. Die Äste des Y-Diagramms stehen für die Sichtweisen. Die Sichtweisen finden ihr Äquivalent immer auf der gleichen Ebene. So befinden sich Gatter auf der Logikebene, welche sich in ihrem Verhalten durch boolesche Gleichungen beschreiben lassen und auf einem Chip physikalisch als Zellen implementiert werden. Im Diagramm erfolgt die Verfeinerung von außen nach innen. Die äußere Schale mit dem größten Radius steht für die höchste Abstraktion. Der Entwurf besteht aus einer Reihe von Transformationen im Y-Diagramm, begleitet durch Verfeinerungs- und Syntheseschritte.

Im idealen Fall kann man konsequent von außen nach innen vorgehen. Jedoch muss man immer wieder validieren und es kann sich herausstellen, dass z. B. ein Entwurfsfehler vorliegt. In einer solchen Situation ist es nötig, auf eine frühere bzw. höhere Ebene zurückzugehen und eine Korrektur vorzunehmen.

Eine ähnliche Vorgehensweise trifft man auch beim Softwareentwurf [190]. Hier ist das Ergebnis zumeist ein fertiges Programm in Maschinensprache.

Die *Systemebene* oder auch *Architectureebene* beschreibt die das Gesamtsystem auf Basis typischer Blöcke wie Speicher, Prozessoren oder Netzwerke. Komplexe Aufgaben und Protokolle charakterisieren diese Ebene. Natürliche Sprachen und Skizzen sind hier als Beschreibungsmittel durchaus nicht selten. Angaben zu konkretem Zeitverhalten oder detaillierter Funktion einzelner Module sind in dieser Stufe nicht zu finden. Geometrisch entsteht schon eine erste Aufteilung der Chipfläche.

Mit der *Algorithmischen Ebene* wird das System durch parallel ablaufende (nebenläufige) Algorithmen beschrieben. Strukturell findet man hier Blöcke, die über Signale miteinander kommunizieren. Elemente der Blöcke sind Funktionen, Prozesse und Kontrollstrukturen, die mittels Operatoren auf Variablen zugreifen [189]. Konkrete Zeitinformationen, wie z. B. die eines Taktsignals, sind hier nicht anzutreffen.

Bei der *Register-Transfer-Ebene* werden Schaltungseigenschaften durch Operationen und Datentransfers zwischen Registern charakterisiert. Das Zeitverhalten ist hier schon so weit verfeinert, dass es Takt- und Setzsignale gibt. Diese Signale lösen durch ihre

Flanken oder Pegel Operationen aus. In der Register-Transfer-Ebene trifft man Elemente wie Addierer, Multiplexer, FSMs oder Register, die über Signale miteinander Daten austauschen.

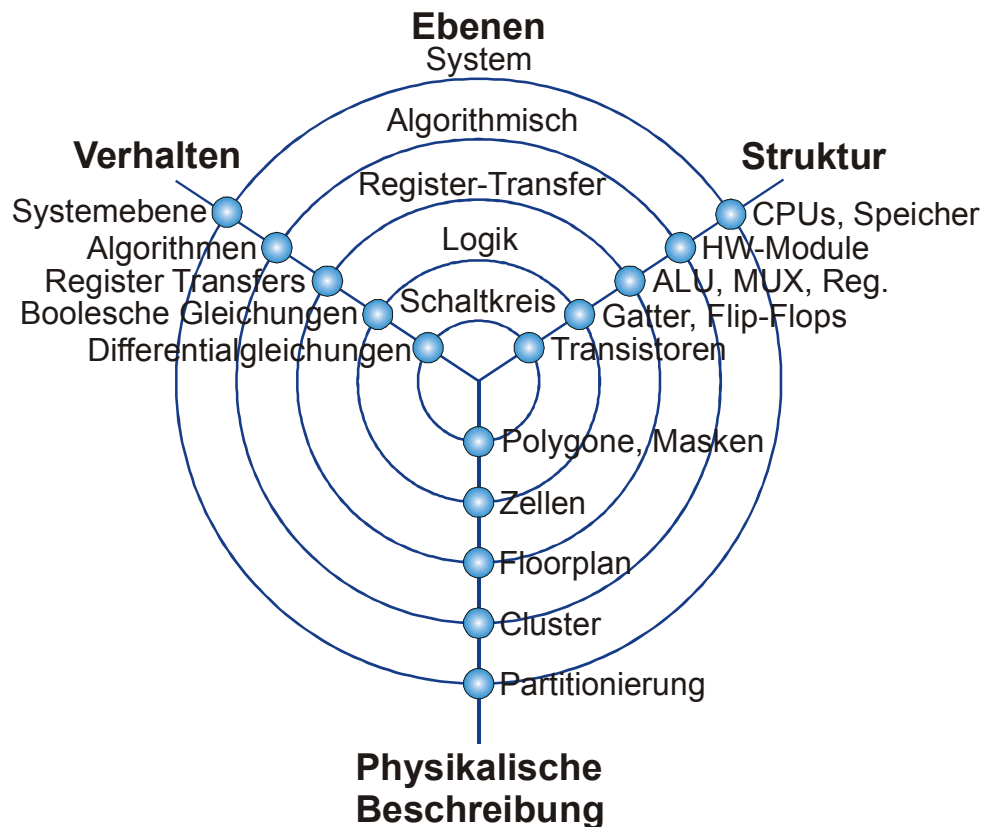


Abbildung 2.1: Y-Diagramm nach Gajski/Walker [191]

Die *Logikebene* beschreibt die Systemeigenschaften in Form logischer Verknüpfungen, welche meist in Form boolescher Ausdrücke vorliegen. Zeitliche Eigenschaften wie Verzögerungszeiten sind bisweilen auch hier anzutreffen. Strukturelle Elemente sind Gatterschaltungen (z. B. AND, OR, Flip-Flops), die man über Signalleitungen miteinander verschaltet. Die Signalverläufe sind wertdiskret und haben typische Logikwerte wie „high“, „low“ oder „unknown“.

Die *Schaltkreisebene* beschreibt das System strukturell als Netzliste. In dieser stehen beispielsweise solche Bauelemente wie Transistoren, Widerstände oder Kondensatoren. Zur Modellierung in dieser Ebene eignen sich meist Differentialgleichungen. Daraus folgt schon für kleine Schaltungen eine zeitaufwändige und speicherintensive Berechnung. Allerdings können die Signale beliebige Werte annehmen und besitzen idealerweise einen zeitkontinuierlichen Verlauf. Geometrisch findet man hier in der Halbleitertechnik Polygone mit verschiedenen Dotierungen.

### 2.2 Einführung zu SystemC

SystemC [29] ist eine Erweiterung von C++ in Form einer Klassenbibliothek. Durch diese Klassenbibliothek lassen sich in C++ Hardware-spezifische Eigenschaften nutzen (→2.4). In das Leben gerufen wurde SystemC 1999 von der Open SystemC Initiative (OSCI). Diese ist eine unabhängige und gemeinnützige Organisation und war damals ein Zusammenschluss verschiedener Firmen aus den Bereichen Halbleiter, EDA, IP und Embedded Software [49]. Die Mitgliedschaft weiterer Firmen, Bildungseinrichtungen, etc. ist möglich und der Unkostenbeitrag richtet sich nach Art der Zugehörigkeit. Vorteil einer Mitgliedschaft ist ein Stimmrecht und so die Möglichkeit einer gezielten Einflussnahme auf die künftige Entwicklung von SystemC. Ziel bei der Gründung von SystemC war es, den Austausch von IP-Kernen zu ermöglichen und eine einheitliche Plattform zur Unterstützung des Hardware/Software-Codesigns zu bekommen. Die SystemC-Bibliothek ist lizenzfrei und nach einer kostenlosen Registrierung in die SystemC-Community unter [29] erhältlich. Der Programmcode selbst liegt offen vor und das Lizenzmodell ist OSI-konform [50]. Mit der gezielten Weiterentwicklung von SystemC beschäftigt sich innerhalb der OSCI eine Reihe von working groups, unterteilt in verschiedene Fachbereiche. Mittlerweile ist SystemC in der IEEE 1666 [42] standardisiert und scheint dadurch noch stärker an Akzeptanz zu gewinnen. Seitdem stieg die Anzahl der unter [29] registrierten Nutzer deutlich an [59].

Die SystemC-Bibliothek setzt sich aus verschiedenen Teilen zusammen, diese sind in der Abbildung 2.2 dargestellt.

Ausgangspunkt und Basis für SystemC ist C++, dies ist in der Abbildung unten dargestellt. Im mittleren Bereich der Abbildung ist die SystemC-Basis-Bibliothek abgebildet.

Grundlage in dieser Basis-Bibliothek ist ein eigener ereignisgesteuerter Simulationskernel. Zur Hardware-Modellierung von Modulen und Verbindungen stehen einige strukturelle Elemente bereit. Zur effektiven Datenverarbeitung gibt es neben den Standard-C++-Datentypen noch weitere Typen, wie zum Beispiel mehrwertige Logik. Einige vorgefertigte *Primitive Channels* in Form von Schablonen erleichtern den Entwurfsprozess.

Neben dieser Basis-Bibliothek gibt es noch einige zusätzliche, vorgefertigte Bibliotheken, welche auf den abstrakteren Ebenen des Systementwurfs verwendet werden. Die *Master/Slave*-Bibliothek bietet schon einen fertigen Rahmen für eine Buskommunikation. Die Verifikations-Bibliothek bietet Unterstützung für Sprachkonzepte der modernen High-Level-Verifikation, wie z. B. die Selbstüberwachung oder die Protokollierung von Transaktionen. Die *TLM*-Bibliothek, derzeit ist die Version 2.0 in Überarbeitung, erlaubt die Zusammenarbeit von Modulen unterschiedlicher Entwurfsebenen.

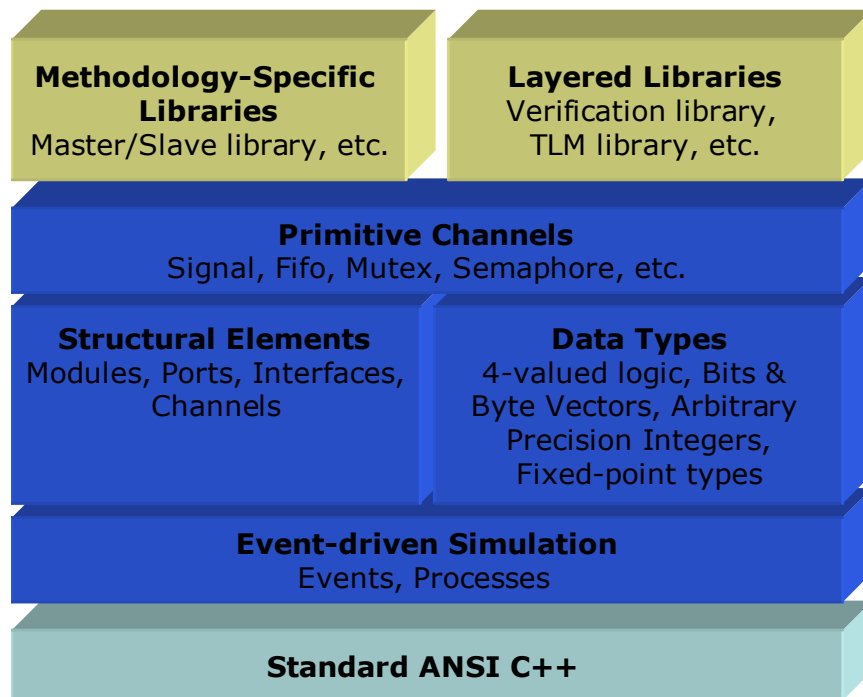


Abbildung 2.2: Architektur der SystemC-Klassenbibliothek

## 2.3 SystemC im Systementwurf

Die Abbildung 2.3 zeigt einen Vergleich eines üblichen HDL-basierten Entwurfsvorgangs (a) und den eines mit SystemC (b). Ausgehend von einer funktionalen Spezifikation kann das Gesamtsystem in C++ beschrieben werden. Ebenso ist es schon möglich, mit der C++-Darstellung durch Kompilieren ein ausführbares Modell zu erhalten. Bei weiterer Verfeinerung blieb allerdings bis zu SystemC, in Ermangelung an Hardware-spezifischen Konstrukten in C++, das Übertragen des Modells in eine HDL nicht aus. Neben zusätzlichen, ausführlichen Kenntnissen und Fähigkeiten in einer zweiten Programmiersprache und weiteren Werkzeugen, die ein solcher Übergang erfordert, führt ein Wechsel oft zu zusätzlichen Fehlern durch Syntaxunterschiede und eventuell sogar auftretenden semantischen Lücken.

In SystemC geschieht der Entwurfsablauf durchgängig in einer Umgebung. Ursprünglich sollte das bis zur Implementierung in Hardware erfolgen. Die Idee war, dass ein Wechsel in eine andere Sprachumgebung entfällt. Dies verkürzt nicht nur die Entwurfszeit und Fehleranfälligkeit, auch die gemeinsame Entwicklung von Hardware und Software wäre so besser unterstützt. Bei einem getrennten Entwurf eines komplexen Systems musste einst der Software-Entwickler erst warten, bis der Hardware-Entwickler fertig war, um detaillierte Programme schreiben zu können. Erst mit einer realistischen Umgebung konnten die Programme überprüft werden. Diese Abhängigkeit führte zu langen Entwicklungszeiten, so dass manche Projekte nie zum

Erfolg gelangen. Mit SystemC können nach der HW/SW-Partitionierung die Hardware und die Programme simultan entworfen werden, denn jeder Entwickler hat ein abstraktes Modell seines Gegenübers mit einer definierten Schnittstelle.

Für eine präzise Hardware/Software-Cosimulation störfte zuweilen die unzureichende Unterstützung des genauen Zeitverhaltens. Jedoch gibt es mittlerweile eine Reihe von Arbeiten, die sich sogar mit der Unterstützung eines Echtzeit-Betriebssystems (RTOS) beschäftigen [54,55].

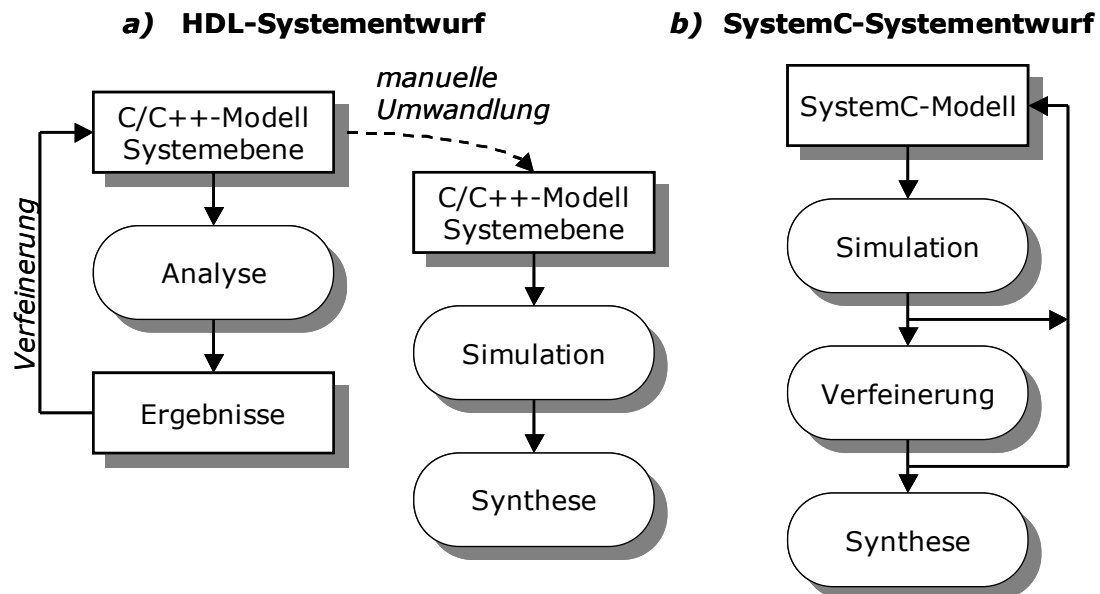


Abbildung 2.3: Vergleich zwischen herkömmlichen HDL- und SystemC-Entwurf

Mit dem Erscheinen von SystemVerilog [178] scheint dieser einheitliche Gesamtansatz wieder an Bedeutung zu verlieren [57,56]. Neben den Konstrukten von Verifikationssprachen bietet SystemVerilog gegenüber früheren HDLs auch eine effiziente Co-Simulation mit SystemC-Blöcken an, so dass hier die Interoperabilität von C++-Beschreibungen auf der System-Ebene und HDL-Darstellungen unterhalb dieser verbessert wurde. Allerdings lassen sich so nicht alle Übertragungsprobleme lösen, z. B. existieren immer noch Sprachunterschiede.

Obwohl SystemC nun auch durchgängig beim Systementwurf verwendet werden kann, bleibt die Einhaltung des Entwurfsebenen-Ansatzes nach Abbildung 2.1 sinnvoll. Die Abbildung 2.4 zeigt die für diese Arbeit relevanten Ebenen.

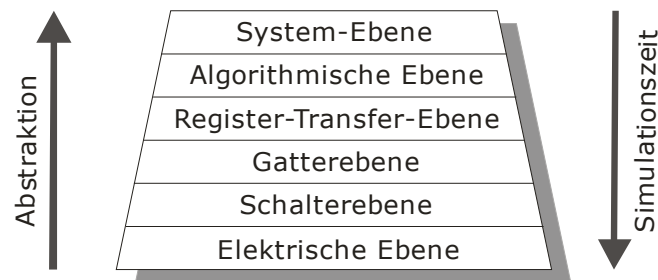


Abbildung 2.4: Ebenen beim durchgängigen Entwurf

Auf der **Systemebene** gibt es bisher keine Standardsprache. Allerdings scheint hier SystemC der De-facto-Standard zu werden. Der Vorteil besteht dabei, dass man eine Spezifikation in ein ausführbares Modell bekommt und sie so schon simulieren kann. Neben SystemC existiert noch UML als universelle Sprache zur Modellierung von Systemen [196]. Jedoch ist dieses Mittel bisher vorwiegend nur für Softwaresysteme geeignet. Mangelnde interne Konsistenz [197], nicht ausreichende Formulierbarkeit von konkretem Objektverhalten, mangelhafte Nebenläufigkeit und fehlende Konstrukte zur Nachbildung des Realzeitverhaltens machen es für eine Hardware-Modellierung nicht attraktiv.

Eine Verfeinerung führt zu **algorithmischen Modellen**. Hier ist der Ansatz des *Transaction Level Modeling* (TLM) sehr verbreitet. Da das Transaction Level Modeling (TLM) derzeit stark an Akzeptanz gewinnt [51,52,60], wird es im Folgenden etwas näher erläutert. Beim TLM-Konzept werden Module zur Kommunikation und funktionale Module unabhängig voneinander entwickelt. Für die Kommunikation zwischen zwei Modulen gilt das gleiche transaktionsbasierte Modell, welches die Funktionalität des Datentransfers betont. Innerhalb des Modells kann nun eine Verfeinerung stattfinden, ohne dass es die anderen Module kümmert. Dies erleichtert einerseits die Entwicklung von komplexen Systemen und führt durch die Abstraktion von Details zu schnelleren Simulationen andererseits. In weiteren Schritten kann dann auch das Kommunikationsmodul verfeinert werden, was zu einer neuen Kommunikationsarchitektur führt. Da Verhalten und Kommunikation beim TLM-Konzept streng voneinander getrennt sind, erlaubt dies die einfache Wiederverwendung von Funktionsblöcken oder die bequeme Implementierung von IP-Kernen. Während andere HDLs wie VHDL oder Verilog nur umständlich das TLM-Konzept übernehmen können, bringt SystemC die nötigen Voraussetzungen hierfür mit.

Neben der Modellierung boolescher/arithmetischer Logik und Registern ist der Entwurf von endlichen Automaten auf der **Register-Transfer-Ebene** in SystemC ebenfalls gut durchführbar. J. Bhasker beschreibt die Entwurfsmethodik dieser Ebene in [37] sehr verständlich. Für SystemC gibt es eine Reihe von Klassen, welche hier die Modellierung erleichtern und die Kompaktheit des Programmcodes verbessern.

Die Elemente der **Gatterebene**, wie die booleschen Grundgatter (Primitives) und Speicherelemente (Flip-Flops, etc.), lassen sich durchaus als Untermenge der Register-Transfer-Ebene bezüglich der praktischen Modellierung betrachten. Daher erlaubt SystemC auch hierzu eine vollständige Modellierung. Jedoch trifft SystemC in diesem Fall auch auf die Kompetenz klassischer HDLs, die jahrelange Erfahrung von HW-Entwicklern und einer besseren Handhabung der Komponenten im Detail, weshalb hier im Entwurf oft andere Beschreibungen als SystemC zum Einsatz kommen. Außerdem bieten viele Werkzeuge der herkömmlichen HDLs eine ausgereifte Entwicklungsumgebung, so dass der Umstieg auf SystemC zusätzlich schwer fällt. Die Synthese von SystemC-Modellen erfolgt bei vielen Werkzeugen schon aus der RT-Ebene [179,180] heraus. Dies ist ein weiterer Grund, weshalb die weitere Systembetrachtung auf der Gatterebene in SystemC entfällt. Selbst TLM-Modelle lassen sich mittlerweile synthetisieren [182]. Die steigende Simulationszeit bei der Modellierung auf der Gatterebene ist ein zusätzliches Problem. Für eine Verbesserung wird dann eine HW-Emulation bevorzugt. Die direkte HW-Implementierung auf einem FPGA sorgt meist für einen Beschleunigungsfaktor von über 100 [38].

Die **Schalterebene** beschreibt, wie die Bezeichnung es schon ausdrückt, das Modell als Schalter. Kennzeichnend ist hier die Möglichkeit des bidirektionalen Signalflusses im Vergleich zur Gatterebene. Außerdem finden sich hier bei einer genauen Modellierung gewisse Verhaltenseigenschaften von Widerständen und Kapazitäten wieder. Diese dienen vornehmlich der Abbildung zusätzlicher Zeitinformationen wie zum Beispiel der von Verzögerungen. Zeitbetrachtungen gibt es allerdings nicht nur auf der Schalterebene, z. B. trifft man diese zuweilen auch auf der Gatterebene. Bedingt durch die Betrachtung von Verzögerungen bekommt man bei einer feinen Auflösung auch einen Einblick bezüglich des Auftretens von Hazards und Glitches.

In der SystemC-Welt sind Arbeiten auf der Schalterebene so gut wie unbekannt. Mit den Möglichkeiten der bidirektionalen Übertragung, der Modellierung von Zeitverhalten in gewissen Grenzen und der Flexibilität die C++ bietet, ist der Einsatz von SystemC grundsätzlich nicht ausgeschlossen.

Die **elektrische Ebene** modelliert das System vornehmlich in Form von Differentialgleichungen. Während bei den bisher erwähnten Ebenen mit digitalen Signalen gearbeitet wurde, mehrwertige Logik inbegriffen, benötigt man hier kontinuierliche Signale. Hier findet der Übergang zu den analogen Signalen statt, wobei u. U. zusätzlich noch digitale Werte verarbeitet werden müssen. Da aber die üblichen Rechner heutzutage fast ausschließlich digital arbeiten, muss bei der Berechnung diskretisiert werden. Diese Diskretisierung ist aber so fein aufgelöst, dass die Ergebnisse hinreichend genau sind. In SystemC ist diese Entwurfsebene wieder interessant. Die SystemC-AMS-Bibliothek [48] bietet hierfür den Einstieg. Innerhalb der OSCI gibt es hierzu eine spezielle Arbeitsgruppe. Ziel dieser ist es nicht, bisherige Analogsimulationen (z. B. PSPICE) zu verdrängen, sondern die Modellierung



heterogener Systeme zu gestatten. Damit wäre eine Co-Simulation von analogen und digitalen Schaltungen, wie sie bei SoCs im Ganzen nötig ist, gegeben.

Neben dem eben vorgestellten Abstraktionsmodell existieren weitere Modelle. So ist in [58] ein Modell dargestellt, welches für die Kommunikation von Bedeutung ist. In der obersten Stufe findet bei der grundsätzlichen Algorithmen-Validierung keine Berücksichtigung der Zeit statt. Mit zunehmender Verfeinerung werden Funktionen in zeitlicher Reihenfolge, dann Buszyklen und später das ganze System zyklengenau modelliert.

Eine ähnliche Berücksichtigung der zeitlichen Abstraktion findet man in [36] mit Fokus auf das TLM.

## 2.4 Modellierung mit SystemC

Mit der Einführung von C++ [176] wurde die Programmiersprache C um eine Reihe von Konstrukten erweitert. Die neuen Sprachfunktionen, welche die objektorientierte Programmierung (OOP) unterstützen, sind Klassen, Templates, das Überladen von Funktionen und Operatoren und die Vererbung von Klasseneigenschaften. Mit diesen Elementen lässt sich eine Applikation in überschaubare Module, den sogenannten Objekten, gliedern. Diese Abstrahierung entspricht auch dem Top-Down-Entwurf bei der HW-Entwicklung [49]. Die Konzepte der objektorientierten Programmierung können wie folgt genutzt werden:

- Mit der Definition von Klassen entstehen gekapselte Module, die ihrerseits Bauelemente enthalten und selbst wieder ein Bauelement sind. Mit ihnen lässt sich der Systemaufbau beschreiben.
- Mit der Vererbung entstehen abgeleitete Klassen, welche die Eigenschaften einer zentralen Basisklasse besitzen. Damit kann man grundlegende Module und Elemente definieren, welche sich je nach Wunsch erweitern und verfeinern lassen.
- Templates gestatten die Definition mit Parametern. Dadurch sind Ports und Signale mit unterschiedlichem Datentyp möglich.
- Das Operator-Überladen erlaubt einen einfacheren und lesbareren Programmcode. Es müssen keine zusätzlichen Funktionsaufrufe definiert und verwendet werden.

In SystemC gilt die gleiche Syntax wie in C++. Es kann die vollständige Funktionalität von C++ auch in SystemC genutzt werden. Zusätzlich bietet SystemC eine Reihe von Konstrukten zur Systembeschreibung. Diese sind nachfolgend aufgeführt:

*Module:* Module sind Container die Prozesse und andere Module enthalten. Mit Modulen wird ein System überschaubar gegliedert, es entstehen üblicherweise Hierarchien. Nach außen wird ein Modul durch seine Interfaces bzw. Ports repräsentiert. Die interne Struktur bleibt so versteckt, was die Einbindung von IP-Blöcken erleichtert.

*Prozesse:* Prozesse beschreiben die Funktionalität innerhalb der Module, wobei diese mit ihrer Arbeitsweise Nebenläufigkeit erlauben. In SystemC gibt es drei verschiedene Ausführungen. METHOD-Prozesse (*SC\_METHOD*) verhalten sich wie Funktionen und werden durch ein Signal aus der Sensitivitätsliste aufgerufen. Nach ihrer Ausführung geben sie die Kontrolle wieder an den Simulator ab. THREAD-Prozesse werden nur einmal gestartet und laufen dann in einer Schleife. Durch *wait()*-Anweisungen kann man sie vorübergehend unterbrechen. Eine Wiederaufnahme des Prozesses erfolgt durch die Aktivität eines Signals, welches in der Sensitivitätsliste steht. Neben dem häufiger verwendeten *SC\_THREAD*-Prozess existiert noch der *SC\_CTHREAD*-Prozess. Dieser arbeitet synchron und sein Ergebnis ist erst mit der nächsten Taktflanke sichtbar. Allerdings wird dieser Prozess in Zukunft wohl nicht mehr unterstützt.

*Signale:* Signale ermöglichen die Verbindung von Modulen und die Aktivierung von Prozessen. Wie in anderen HW-Beschreibungssprachen kann die Zuweisung verzögert erfolgen. Typischerweise ist ein Signal durch einen zugewiesenen Datentyp charakterisiert. An einem Signal können auch mehrere Treiber angeschlossen sein. Hierzu muss lediglich der aufgelöste Signaltyp *sc\_signal\_resolved* verwendet werden.

*Ports:* Ports bilden die Schnittstelle eines Moduls. Diese werden mit den Signalen verbunden. Es gibt drei Arten von Ports: Eingänge, Ausgängen und bidirektionale.

*Datentypen:* Zusätzlich zu den schon in C++ vorhandenen Datentypen bietet SystemC weitere. Dies sind Fixkommatypen, ganzzahlige Typen mit einstellbarer Länge und zwei-/vierwertige Logiktypen. Diese Datentypen unterstützen die Modellierung auf unterschiedlichen Abstraktionsebenen.

*Clocks:* Clocks sind spezielle Signale, die der Zeitfortschaltung der Simulation dienen. Es dürfen mehrere Clocks mit zueinander unterschiedlichen Phasenbeziehungen definiert werden.

*Reactivity:* In SystemC werden Mechanismen für das Warten auf Taktflanken, Ereignissen, Signalübergängen und dem Überwachen von festgelegten Signalen unterstützt.

*Debugging und Tracing:* Die SystemC-Klassen besitzen eine Laufzeitüberwachung, welche zum Beispiel nicht angeschlossene Ports meldet. Mit dem Tracing können Signalkurven in ihrem Verlauf in Dateien gespeichert werden. Die verwendeten Formate sind VCD, WIF und ISDB, sie lassen in geeigneten Programmen eine graphische Betrachtung zu.

*Systemkonzepte:* Hier werden die Beschreibungen von Kommunikationskanälen und abstrakten Kommunikationsprotokollen ermöglicht.

## 2.5 SystemC-Projektumgebung

Die Entwicklung eines SystemC-Projektes erfolgt nahezu immer in einem ähnlichen Schema. Kernstück des Entwurfs ist eine C++-Entwicklungsumgebung. Diese besteht typischerweise aus einem komfortablen Editor, einem Compiler und einem Debugger. Dies ist in der Abbildung 2.5 in der oberen Hälfte angedeutet. Das SystemC-Design liegt hier wie die zusätzliche Testbench in Form von Quelltexten vor. Zusammen mit der vorkompilierten SystemC-Bibliothek erzeugt der Compiler hieraus eine ausführbare Datei. Dieses Programm ist dann die komplette Simulationsumgebung. In der Abbildung 2.5 ist dies der mittlere untere Teil. Kern ist hier das ursprüngliche Design bzw. das Device under Test (DUT). Das DUT wird durch eine Stimuli-Komponente mit Eingangsdaten versorgt und so angeregt. Die Eingabedaten können entweder aus einer externen Datei stammen oder auch schon im Kompiliervorgang in das Projekt mit eingebunden worden sein. Ein Monitor-Modul nimmt die Ausgabedaten entgegen, bereitet sie auf und speichert diese zweckentsprechend. Neben einer typischen Ausgabe in einer Text- oder Binärdatei bietet SystemC schon die Möglichkeit zusätzliche Waveform-Dateien anzulegen. Mit geeigneten Programmen und diesen Dateien können die Signal- und Datenwerte der Dateien, die während der Simulation aufgezeichnet wurden, graphisch ausgewertet werden. Außerdem existiert noch eine Ausgabe des Simulationsfortschritts auf dem Bildschirm.

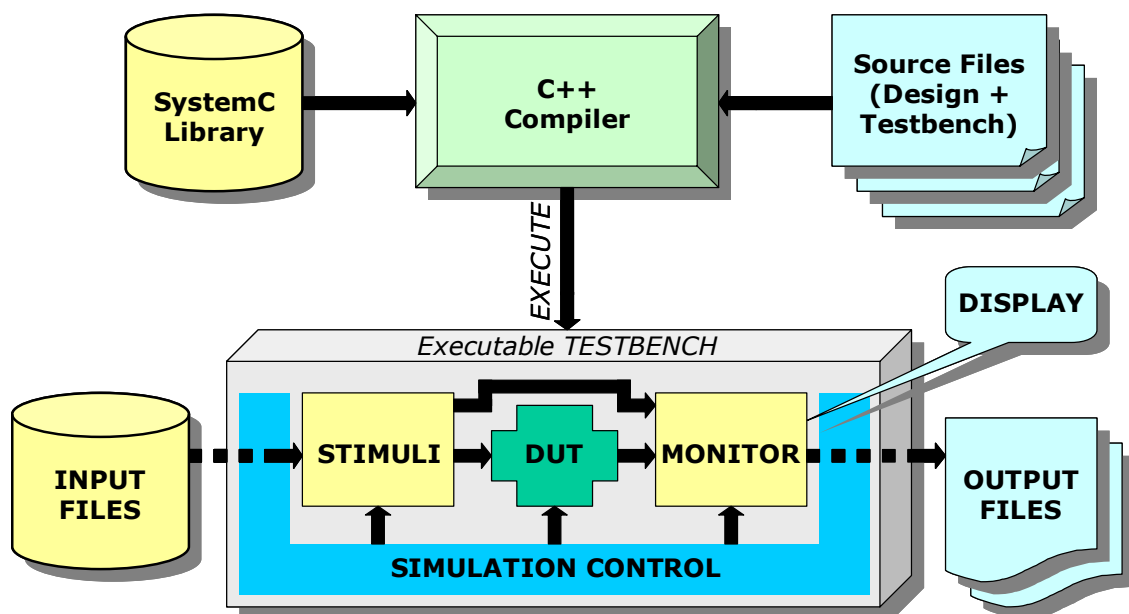


Abbildung 2.5: SystemC-Projektumgebung

Soll ein SystemC-Projekt für einen HW-Syntheseschritt genutzt werden, so wird normalerweise nur der Quelltext des Designs dem Synthese-Werkzeug zugeführt. Das Ergebnis des Vorgangs ist dann beispielsweise eine Datei, die einem FPGA-Tool gegeben wird, welches im weiteren Verlauf die ursprüngliche SystemC-Beschreibung in Hardware überführt.

## 3 Simulation

Zunehmende Chipflächen und eine wachsende Integrationsdichte machen den rechnergestützten Schaltungsentwurf unumgänglich. Um die zunehmende Komplexität zu beherrschen, wird ein mögliches Modell eines Entwurfs in verschiedenen Abstraktionsebenen behandelt (→Abb. 2.1).

Wegen vielfältiger möglicher Fehlerfälle ist es nötig sicherzustellen, dass das verwendete Modell korrekt ist. Hierbei wird überwiegend die Implementierung gegen ihre Spezifikation geprüft.

**Verifikation** ist der Nachweis, dass ein erstelltes Modell das in der Spezifikation beschriebene Verhalten hat [1].

Jedoch ist die formale Verifikation zur Entwurfsüberprüfung oftmals schwierig. Bei großen Modellen scheitert diese einfach an der Komplexität. Für Teilschaltungen wird sie durchaus erfolgreich eingesetzt.

Oftmals wird der Begriff der Verifikation im Zusammenhang mit der Simulation gebracht. Aber das im Begriff der Verifikation enthaltene Versprechen eines Wahrheitsbeweises wird bei der Simulation nicht erfüllt. Die Simulation kann Schaltungsfehler finden, aber nicht beweisen, dass die Schaltung fehlerfrei ist. Besser ist es, man spricht im Zusammenhang mit Simulationen von einer Validierung.

**Validierung** oder **Validation** ist die Prüfung eines Modells gegen ihre Spezifikation, die mit einer Bestätigung oder Widerlegung endet.

Eine Validierung kann sowohl eine Simulation als auch eine Verifikation sein. Eine Validierung einer Schaltung lässt sich auch so durchführen, dass man das Modell gleich in Hardware umsetzt und dann ihr Verhalten analysiert. Allerdings kann dies durch die anzuschaffende Hardware und den nötigen Syntheseschritten teuer werden.

Die Simulation erfolgt als auszuführende Software auf einem Rechner. Hierzu wird typischerweise das vorliegende Modell in ein Programm eingelesen und ausgeführt.

Bei der Simulation werden Eingangsmuster eingelesen und die sich ergebenden Ausgangsmuster aufgezeichnet und analysiert. Die Analyse der Ausgangsdaten ist typischerweise ein Vergleich mit Referenzdaten, welche z. B. aus der Spezifikation oder bei einer Entwurfsverfeinerung aus dem vorhergehenden Modell bekannt sind. Ergeben sich nun Unterschiede beim Vergleich, so kann von einem inkorrekten Modell ausgegangen werden. Allerdings findet man auf diese Weise keine Entwurfsfehler im Referenzmodell. Vorteile der Simulation sind die gute Beobachtbarkeit des Modells, die flexible Änderbarkeit und die virtuelle Modellierung ohne zusätzlichen Hardware-Aufwand. Nachteilig ist die manchmal deutlich langsamere Ausführung gegenüber einer direkten Hardware-Implementierung oder der Bedarf an leistungsfähigen Rechnern zur

Simulationsausführung. Die Simulation selbst kann auf allen Ebenen des Entwurfs stattfinden. Fast immer besteht das Problem, ein Modell in einer vertretbaren Zeit zu validieren. Hier ist die Abwägung einer schnellen Simulation gegen eine sich verringernde Genauigkeit nötig. Mit höheren Abstraktionen erreicht man meistens mehr Geschwindigkeit und mit Modellen auf unteren Ebenen des Entwurfs mehr Genauigkeit. Eine brauchbare Lösung stellt hier die Partitionierung des Systems und einer daraus möglichen gemischten Simulation (*mixed mode simulation*) dar. Da sich die vorliegende Arbeit hauptsächlich mit dem Entwurf auf der Register-Transfer-Ebene und unterhalb dieser bewegt, erfolgt im Folgenden zur Einführung und Standortbestimmung ein Überblick zur Logiksimulation und anschließend zur Fehlersimulation.

### 3.1 Logiksimulation

Die Abbildung 3.1 zeigt eine mögliche Einteilung der Logik-Simulationsverfahren. Zu den schon seit langem genutzten Verfahren der kompilierten und interpretierenden Simulationen gewinnt in zunehmenden Maße die native-kompilierte Technik, insbesondere in modernen VHDL- und Verilog-Simulatoren, an Bedeutung. Außerdem lassen sich die Simulationen noch in einer sequenziellen oder parallelen Abarbeitung unterscheiden. Eine parallele Simulation verfolgt zumeist den Zweck einer Simulationsbeschleunigung ( $\rightarrow 7$ ).

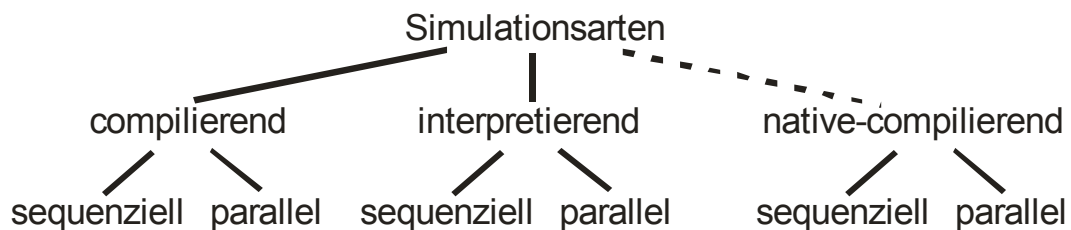


Abbildung 3.1: Übersicht von Arten der Logiksimulation

#### 3.1.1 Kompilierte Simulation

Kompilierte Simulationen, auch als übersetzende Verfahren oder *streamline code simulation* bezeichnet, überführen die Schaltung in ein System boolescher Gleichungen. Dazu werden die Gatter entsprechend ihrem Abstand von den primären Eingängen geordnet (engl. *circuit levelizing*). Schaltungsrückkopplungen werden wie primäre Eingänge behandelt. Im nächsten Schritt erzeugt man gemäß der geschaffenen Ordnung ein Programm, welches die Werte berechnet [1]. Es entsteht direkter Maschinencode und somit ein ausführbares Programm. Schaltungsbeschreibung und Simulator sind eine Datei. Kompilierte Simulatoren haben eine hohe Ausführungsgeschwindigkeit, jedoch erfordert die kleinste Modelländerung ein erneutes Kompilieren. Dabei ist die

Kompilierzeit eine oftmals nicht zu vernachlässigende Größe im zu betrachtenden Simulationszyklus. Zudem lassen sich Timing-Informationen nicht berücksichtigen. Für viele kompilierten Simulationen ist es typisch, dass sie äquitemporal arbeiten. Es werden also alle Elemente bei der Simulation berücksichtigt, obwohl dies vielleicht unnötig ist, da für viele Elemente keine Änderung eintritt.

### 3.1.2 Interpretierende Simulation

Bei der interpretierenden Simulation ist die Datenstruktur der Schaltung in eine spezielle Form codiert, die dann der Logiksimulator ausführen kann. Meistens verfügt ein interpretierender Simulator über eine Benutzungsschnittstelle oder Bedienoberfläche, die es dem Bediener ermöglicht, die Simulation interaktiv zu beeinflussen. So ist es beispielsweise erlaubt, Zustände und Parameter zu ändern. Durch das Verwenden von Kommando-Dateien lassen sich hier Simulationsszenarien wiederholen und automatisieren. Für viele interpretierenden Simulatoren ist typisch, dass sie ereignisgesteuert arbeiten. Das heißt, es werden nur die Elemente simuliert, bei denen ein Ereignis vorliegt und bei denen es somit nötig ist.

### 3.1.3 Native-code-compiled Simulation

Dieses Verfahren kann als Kombination der kompilierten und der interpretierenden Simulation betrachtet werden. Hierbei wird die Schaltungsbeschreibung in einen Objekt-Code des Simulators kompiliert. Dabei werden die Simulationsvorbereitung und der Simulationsablauf optimiert [10]. Auf diesen Weg lässt sich beispielsweise ein Schaltungsmodell als eine Mischung aus ereignisgesteuerter und zyklenbasierter Simulation ausführen. Viele moderne HDL-Simulatoren nutzen dieses Verfahren [24,25].

Neben der *native compilation*, trifft man gelegentlich auf ein ähnliches Verfahren, nämlich die *transcompilation* [203]. Bei diesem Weg wird die Hardwarebeschreibung in eine Programmiersprache transformiert. Typischerweise ist dies C++ [189]. Anschließend wird der Quellcode in ein ausführbares Programm des Simulationsrechners kompiliert. Zwei Vorteile sind hier von herausragender Bedeutung. Zum einen ist die Abbildung in eine Programmiersprache universeller und Plattform-unabhängiger als ein proprietäres Format bei der *native compilation*. Andererseits lässt sich die jahrelange Erfahrung in der Entwicklung eines Kompilers nutzen, der einen hochoptimalen Code erzeugt.

### 3.1.4 Tabellengesteuerte Simulation

Neben den oben vorgestellten Simulationsverfahren wird in der Literatur [3] oft die tabellengesteuerte Simulation vorgestellt. Meist nutzen interpretierende Simulationen die tabellengesteuerte Technik. Da aber in jüngerer Zeit auch kompilierte Verfahren Tabellen verwenden [5,29], macht eine separate Betrachtung durchaus Sinn.

Bei dieser Simulationsform ist das Verhalten der Bauelemente in Tabellenform abgelegt. Zum Beispiel kann man ein AND-Gatter auch über eine Wahrheitstabelle definieren. Neben der implizit im Rechner vorliegenden zweiwertigen Logik lässt sich bei einer Tabelle auch mehrwertige Logik implementieren [30]. Zusätzlich erlaubt die Tabellenform die Aufnahme von Timing-Informationen. So lassen sich auch Informationen über Verzögerungszeiten, Flankenanstiegs- und -abfallzeiten nachbilden. Die Schaltungsbeschreibung selbst kann man auch in Tabellenform abbilden. Hierbei ähnelt der Aufbau der einer Gatternetzliste, deren Umsetzung in Struktur und Aufbau stark abhängig vom verwendeten Simulator ist.

### 3.1.5 Simulationstypen

Je nach Abarbeitung der Elementfunktionen kann zwischen einer kompletten Abarbeitung, der ereignisgesteuerten Simulation und der zyklenbasierten Simulation unterschieden werden. Die komplette Abarbeitung war eine der ersten Simulationen, ist sehr einfach und wurde schon im Abschnitt kompilierte Simulation (→3.1.1) erwähnt. Die ereignisgesteuerte und die zyklenbasierte Variante erfordern eine komplexere Simulationssteuerung.

#### 3.1.5.1 Ereignisgesteuerte Simulation

Da sich bei einer realen Schaltung sehr selten Änderungen für alle Schaltungsteile ergeben, ist eine komplette Berechnung aller Elemente überflüssig. Typischerweise liegt die Aktivität in einer Schaltung selten über 5 % bis 10 %. Um hier einen Effektivitätszuwachs zu erreichen, wird oft das Ereignisprinzip eingesetzt. Als ein Ereignis wird dabei die Signaländerung an einem Schaltungsknoten bezeichnet. Man simuliert nur die Elemente, für die sich zu einem diskreten Zeitpunkt eine Signaländerung an einem Eingang ergibt. Die Signaländerungen verfolgt man über Einträge in einer Ereigniswarteschlange. Für jedes Signal kommt es zu einer Berücksichtigung von Wert und Zeit der Änderung. Zu einem Simulationszeitpunkt werden zu den Ereignissen in der Warteschlange die zugehörigen Aktionen ausgelöst und deren Ergebnisse wieder in der Warteschlange gespeichert [4]. Die Abarbeitung einer Ereignisgruppe in einem diskreten Zeitpunkt wird oft als Delta-Zyklus bezeichnet



(auch  $\Delta$ -Zyklus). Zu einem Zeitpunkt können mehrere Berechnungen nötig sein. In der Abbildung 3.2 Wird dies verdeutlicht.

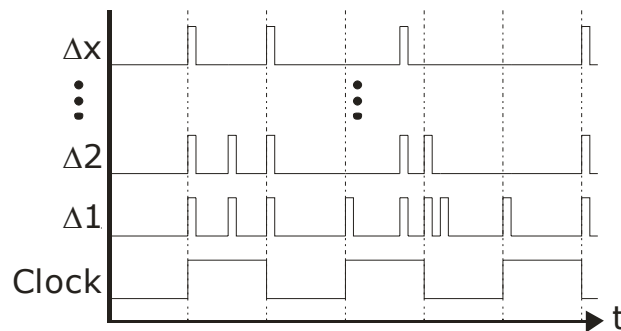


Abbildung 3.2: Simulationszyklen bei der ereignisgesteuerten Simulation

Sind alle Ereignisse in der Warteschlange abgearbeitet und ist das Verhalten stabil ( $\Delta$ -Zyklus abgeschlossen), so wird weiter auf den nächsten Ereigniszeitpunkt gesetzt. Diese Art der Simulation ist sehr flexibel und erlaubt sowohl nahezu beliebiges Zeitverhalten als auch erweiterte Signalwerte. Allerdings ist die ereignisgesteuerte Simulation aber aufwändig im Finden aktiver Elemente und dem Verwalten der Ereignisse. Bei großer Schaltungsaktivität kann die Rechenzeit rapide steigen.

### 3.1.5.2 Zyklusbasierte Simulation

Die zyklusbasierte oder auch zyklusbasierte Methode macht eine Einschränkung hinsichtlich der Simulationszeitpunkte. Anstatt jede Signaländerung zu beliebigen Zeitpunkten zu berechnen, werden Signaländerungen nur zu den Taktzeitpunkten in richtiger Reihenfolge ermittelt.

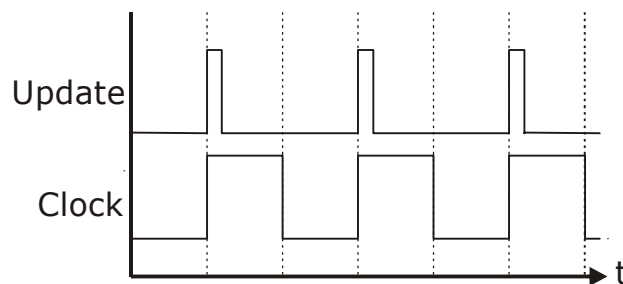


Abbildung 3.3: Simulationszyklen bei der zyklusbasierten Simulation

Die Abbildung 3.3 verdeutlicht die zyklische Aktualisierung der Simulation zur positiven Flanke eines Taktes. Ergebnis ist eine schnellere Ausführungszeit und ein geringerer Speicherbedarf im Vergleich zur ereignisgesteuerten Simulation [23].

Restriktion dieser Variante ist allerdings die Einschränkung auf taktsynchrone Schaltungen. Ein Ausweg aus dieser Beschränkung liegt in der Partitionierung des

Modells in synchrone und asynchrone Teile. Die synchronen Teile werden dann zyklusbasiert und der Rest ereignisgesteuert simuliert.

### 3.1.6 SystemC-Simulationskernel

Für den Simulationsablauf in SystemC sind einige C++-Klassen zuständig, diese ergeben den sogenannten Simulationskernel. Dieser Teil ist in der SystemC-Bibliothek und bildet die Grundlage für ein ausführbares SystemC-Projekt. Der Simulator selbst kann als eine Komposition der oben dargestellten Simulationsverfahren und -typen betrachtet werden. Die komplette Simulation, also Schaltung und Simulationssteuerung, ist ein fertiges Programm in Maschinencode wie bei der kompilierten Simulation. Die Berechnung erfolgt jedoch nicht komplett für alle Elemente, wie bei der klassischen Compiler-gesteuerten Simulation, sondern vorzugsweise ereignisgesteuert. Es lassen sich auch Timinginformationen definieren. Die erste veröffentlichte SystemC-Version 0.9 war allerdings noch rein zyklusbasiert. In der Literatur findet man für SystemC sowohl den Begriff der zyklusbasierten (cycle based) [39] als auch den Begriff einer ereignisgesteuerten Simulation (event driven) [37], was für etwas Verwirrung sorgt. Letzten Endes entscheidet das Model of Computation (MoC) [31] bzw. die Art der Projektimplementierung [53], ob ein schnelle zyklusbasierte oder eine genauere ereignisgesteuerte Simulation läuft.

Die Reihenfolge der Abarbeitung bei der Simulation wird in SystemC durch den objektorientierten Ansatz explizit durch die Methodenaufrufe und implizit durch das Berechnungsmodell bestimmt.

Der SystemC-Scheduler arbeitet nach dem folgenden Algorithmus [37,40]:

1. Initialisierungsphase – alle Prozesse (außer SC\_CTHREADS) werden in einer nicht spezifizierten Reihenfolge ausgeführt.
2. Evaluationsphase – es wird ein Prozess ausgewählt, welcher bereit ist, seine Ausführung zu starten bzw. wieder aufzunehmen. Dies kann unmittelbare Ereignismeldungen nach sich ziehen, die weitere Prozesse veranlassen in der gleichen Phase zu starten.
3. Gibt es noch Prozesse, die ausgeführt werden müssen - dann wird Schritt 2 wiederholt.
4. Aktualisierungsphase – es werden nun alle schwebenden Aufrufe mit *update()* ausgeführt für die ein *request\_update()*-Aufruf aus Schritt 2 vorliegt.
5. Gibt es eine verzögerte Meldung (notification), so wird bestimmt, welcher Prozess bezüglich dieser Verzögerung auszuführen ist und zu Schritt 2 gegangen.
6. Sind keine zeitlichen Meldungen mehr vorhanden, ist diese Simulation beendet.

7. Ansonsten wird die momentane Simulationszeit auf das nächstmögliche Ereignis erhöht.
8. Es wird bestimmt, welche Prozesse mit einer schwebenden Meldung zum jetzigen Zeitpunkt bereit für eine Ausführung sind und es wird wieder zu Schritt 2 gegangen.

## 3.2 Fehlersimulation

Grundsätzlich geht man bei der Fehlersimulation so vor, dass in einer Schaltung ein Fehler, basierend auf einem Fehlermodell (→4.3), eingebaut und dann die Schaltung mit diesem Fehler und mindestens einem Eingabemuster simuliert wird. Ergibt sich ein Unterschied zwischen dem korrekten Ausgangsmuster und der Ausgabe mit eingebautem Fehler, so gilt der Fehler mit dem verwendeten Eingabemuster als erkannt. Die Abbildung 3.4 zeigt den Algorithmus für eine einfache Fehlersimulation. Das Verhältnis der durch die Folge an Eingabemuster gefundenen Fehler zur Gesamtzahl der modellierten Fehler wird als Fehlererkennungsgrad oder Fehlerüberdeckung  $FÜ$  bezeichnet [7].

$$FÜ = \frac{\text{Anzahl erkannte Fehler}}{\text{Anzahl modellierte Fehler}} \quad (3.1)$$

Die Fehlersimulation ist auch ein quantitatives Hilfsmittel, um Testsätze zu beurteilen. Testsätze sind Sequenzen von Eingabemustern bzw. Testmustern. Der Rechenaufwand ist bei der Fehlersimulation von drei Faktoren abhängig:

- Schaltungsgröße,
- Anzahl der Fehlerliste,
- Anzahl der Muster.

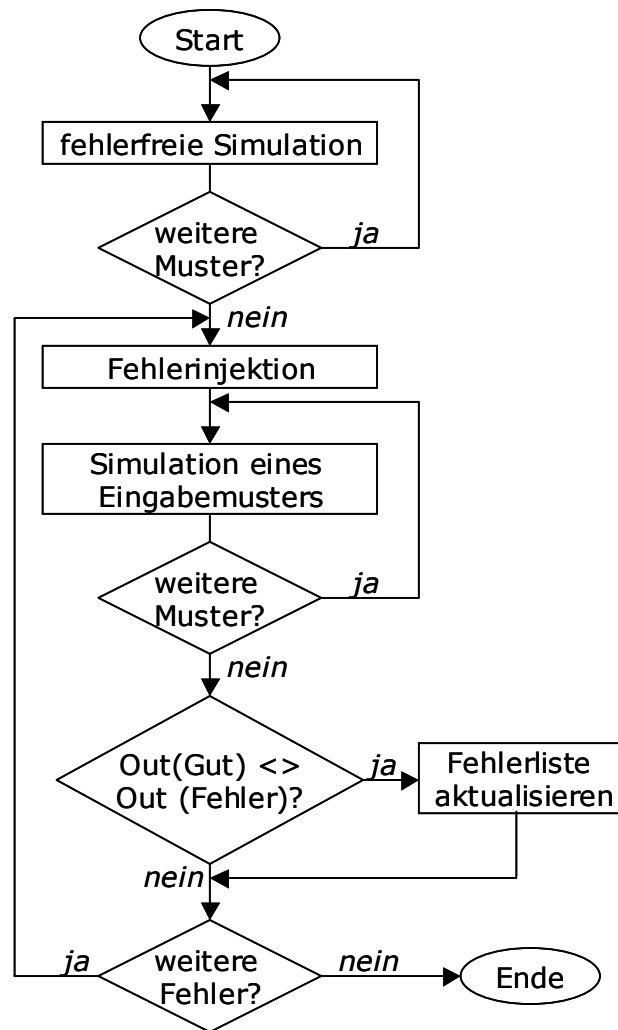


Abbildung 3.4: Ablauf einer einfachen Fehlersimulation

In der Praxis wurde in Abhängigkeit von der Schaltungsgröße  $N$  eine Zunahme der Rechenzeit in Höhe von  $O(N^2)$  bis  $O(N^3)$  festgestellt [9,5]. Die Abbildung 3.5 verdeutlicht den Rechenaufwand.

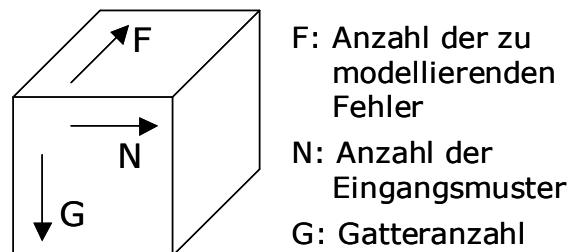


Abbildung 3.5: Rechenaufwand bei einer kombinatorischen Schaltung

Ein bedeutender Einsatzzweck der Fehlersimulation digitaler Schaltungen findet man in der Automatischen Test Pattern Generierung (ATPG). Hier wird die Effizienz der

ATPG wesentlich von der Qualität des Fehlersimulators beeinflusst. Üblicherweise kann man für ein ermitteltes Testmuster durch die Fehlersimulation bestimmen, welche Fehler zusätzlich erkannt werden. In diesem Fall können die zusätzlich erkannten Fehler aus der Liste der noch vom Muster-Generierungs-Tool zu bearbeitenden Fehler gestrichen werden. Häufig anzutreffen ist für eine Fehlersimulation in der ATPG die Einzelfehlerannahme. Hier wird zu einem konkreten Zeitpunkt nur ein Fehlerfall betrachtet. Mehrfachfehlerinjektionen führen einerseits durch ihre umfangreichen Kombinationsmöglichkeiten schnell zu einem nicht zu bewältigenden Rechenaufwand. Andererseits genügt ein Fehler, um ein Bauteil als defekt zu kennzeichnen. Für die ATPG lassen sich drei wesentliche Anwendungen für die Fehlersimulation spezifizieren [5]:

- Validierung der Testmenge: Aus der Entwurfphase liegen häufig schon Eingangsmuster vor. Mit der Fehlersimulation lassen sich diese Muster auch bezüglich einer Fehlererfassung untersuchen und für einen späteren Test verwenden.
- Minimierung der Testmengen: Wird ein Fehler durch mehrere Muster erkannt, können überflüssige Testmuster entfernt werden.
- Beschleunigung der Testerzeugung: Leicht erkennbare Fehler werden durch Zufallsmuster schnell gefunden [9,13]. Die Fehlersimulation ermittelt hier, welche das sind. Nur für die verbleibenden „harten“ Fehler muss dann die rechenaufwändigere Testerzeugung benutzt werden.

Ein anderer Einsatzbereich ergibt sich für die Fehlersimulation bei der Analyse bzw. Diagnose einer Fehlererscheinung. Während es bei der ATPG noch darum ging, mit einem Testmuster möglichst viele Fehler zu erfassen, ist es bei der Fehlerdiagnose wichtig ein Muster zu haben, was einem Fehler eindeutig zugeordnet werden kann. Gegebenenfalls ist es aber auch möglich, mit mehreren Mustern eine Fehlerortung zu erreichen. Findet beispielsweise ein Muster M1 die Fehler sa1(A) und sa1(B) und ein Muster M2 die Fehler sa1(A) und sa0(C), so kann durch die Auswertung beider Muster der Fehler sa1(A) gefunden werden.

Solch eine Fehlerdiagnose findet man beispielsweise in der Qualitätssicherung während des Produktionsablaufs oder bei der Überführung eines neuen Produktes in die Serienfertigung. Häufen sich hier die Ausschussquoten, muss gezielt nach der Fehlerursache gesucht werden, um die Effektivität zu steigern. Eine Fehlerhäufung an einem bestimmten Ort in der Schaltung kann z. B. durch einen Schwachpunkt im Design oder auch durch einen nicht optimalen Technologieschritt entstanden sein.

Eine zunehmende Bedeutung bekommt die Fehlersimulation auch für die Analyse des Schaltungsverhaltens im späteren Einsatz. Durch die Injektion transienter Fehler wird hier beispielsweise die implementierte Fehlertoleranz [151] validiert. Hierbei kann durchaus der Wunsch nach Analyse von Mehrfachfehlern bestehen. Kommt es nämlich

zu einer Störeinwirkung von außen, ist es sehr wahrscheinlich, dass mehr als ein Fehler in der Schaltung auftritt. Außerdem sind solche Systeme im Feld selten homogen, z. B. SoCs. Hier arbeiten verschiedene Komponenten wie Prozessoren, Speicher, Busse, analoge Baugruppen, etc. zusammen. Ein klassischer Gate-Level-Fehlersimulator mit Einzelfehlerbetrachtung ist hierzu ungeeignet. Deshalb greift man aufgrund der nötigen Flexibilität wieder auf leistungsfähige HDL-Simulatoren wie z. B. [24] oder Mixed-Signal-Simulatoren [27] zurück. Diese analytischen Fehlersimulationen zur Nachahmung des späteren Einsatzes werden zunehmend wichtig. Bedingt durch immer kleiner werdende Strukturen und niedrigeren Spannungspegeln beim Chipdesign, werden die Schaltungen empfindlicher gegen sporadische Störeinflüsse von außen, verursacht durch elektromagnetische Strahlung oder Partikelstrahlung. Die Validierung von Mechanismen der Fehlertoleranz wird hier notwendig.

### 3.2.1 Serielle Fehlersimulation

Diese Form der Fehlersimulation ist die ursprüngliche und einfachste. Sie wird auch als *single pattern single fault simulation* (SPSF) bezeichnet. Hier wird zunächst eine Simulation der fehlerfreien Schaltung (Gutsimulation) durchgeführt und anschließend die Schaltung durch den Einbau eines Fehlers modifiziert und wieder simuliert. Danach folgen die nächsten Schaltungsmodifikationen und Simulationen, der Vorgang verläuft seriell. Die Ergebnisse jeder Simulation werden gespeichert und zur Bewertung am Ende auf Unterschiede gegen die Gutsimulation verglichen. Der Vorteil dieser Variante liegt darin, einen gewöhnlichen Logiksimulator zu verwenden. Jedoch entsteht oft ein beträchtlicher Rechenaufwand. Es folgen einige Betrachtungen in Anlehnung an [7] zum Rechenzeitaufwand ohne Zuhilfenahme spezieller Beschleunigungsalgorithmen. Geht man von einer kombinatorischen Schaltung mit einer Schaltungsgröße von  $N$  Gattern und einer durchschnittlichen Rechenzeit  $T_G$  pro Gatter aus, so ergibt sich eine Simulationszeit ( $T_S$ ) von:

$$T_S = N \cdot T_G \quad (3.2)$$

Erfolgt nun eine Einzelfehlerbetrachtung mit  $F$  Fehlerpunkten pro Gatter und eine Fehlerbetrachtung über alle Gatter, dann folgt eine Fehlersimulationszeit  $T_{FS}$  von:

$$T_{FS} = (F \cdot N^2 + N) \cdot T_G \quad (3.3)$$

Mit Berücksichtigung der Testsatzlänge  $L$  ergibt sich somit eine Gesamtlaufzeit  $T_{FSL}$  von:

$$T_{FSL} = (F \cdot N^2 + N) \cdot T_G \cdot L \quad (3.4)$$

Eine ähnliche Gleichung lässt sich statt der Gatterbetrachtung auch für die Knoten bzw. Leitungen einer Schaltungen herleiten. Werden eine Testsatzlänge  $L$ , eine Fehleranzahl  $F$  pro Knoten  $KN$  und eine durchschnittliche Rechenzeit  $T_S$  zugrunde gelegt, dann ergibt sich eine Gesamtrechenzeit von:

$$T_{FSL} = (F \cdot KN + 1) \cdot T_S \cdot L \quad (3.5)$$

Für eine Beispielschaltung von  $N = 100000$  Gattern, einer Gatterrechenzeit  $T_G = 1$  ns und bei  $F = 3$  angenommenen Fehlern pro Gatter ergibt sich eine Zeit  $T_S = 0,1$  ms für die einfache Logiksimulation ( $\rightarrow 3.1$ ) und eine Fehlersimulationszeit  $T_{FS} \approx 30$  s ( $\rightarrow 3.2$ ) bei nur einem Simulationszyklus! In Anlehnung an [6] würde man für diese Schaltung ca. 10000 Testmuster brauchen. Dies ergäbe dann eine Gesamtrechenzeit von 3,5 Tagen. Hier zeigt sich, dass die serielle Fehlersimulation schnell an die Grenzen des zeitlich Sinnvollen und Machbaren stößt.

Eine gewisse Verringerung des Rechenaufwandes erreicht man, wenn zunächst die fehlerfreie Variante berechnet wird, anschließend eine fehlerhafte und man beim Vergleich der Ausgangsmuster beim ersten Unterschied abbricht. Diese Vorgehensweise wird als *fault dropping* bezeichnet und eignet sich beispielsweise gut für die ATPG. Abhängig von der Schaltung kann der Aufwand so bis zu 90 % reduziert werden [7]. Bei Fehleranalyseverfahren oder einer umfassenden Musterbetrachtung ist das *fault dropping* eher ungeeignet, da man nur das erste Fehler provozierende Eingangsmuster kennt. Ein Fehlerlexikon, das Informationen darüber enthält, welche Muster welche Fehler aufdecken, lässt sich so nicht erstellen.

### 3.2.2 Parallele Fehlersimulation

Die parallele Fehlersimulation ist eine schon recht frühe Variante um den Rechenaufwand zu verringern [63]. Bei dieser Technik werden in einem Maschinenwort mehrere Muster mittels Nutzung Bit-paralleler Prozessorbefehle gleichzeitig berechnet. Der Prozessor verwendet bei einer Operation immer alle Bits eines Wortes, also liegt es nahe, auch alle zu nutzen. Jedes Bit ist dabei unabhängig vom Nachbar-Bit und ist Teil eines Musters. Dieses Verfahren lässt sich besonders gut auf der Gatterebene anwenden, da die logischen Grundoperationen typischerweise als Befehle im Prozessor schon implementiert sind. Solch eine Vorgehensweise wurde schon bei den ersten kompilierten Simulationen zum Teil angewendet. Hier ergibt sich theoretisch eine Steigerung der Simulationsgeschwindigkeit proportional zur Länge des Maschinenwortes. Eigene Experimente haben gezeigt, dass bei ereignisgesteuerten Simulationen sich weniger Geschwindigkeitsvorteile als bei der kompilierten Simulation ergeben. Dies liegt darin begründet, dass die Wahrscheinlichkeit für eine Änderung eines Maschinenwortes zum Vorgängerwort durch die Aufnahme mehrerer Muster deutlich höher ist als bei der Einzelmustersimulation. Da die Ereignissteuerung

auf mindestens einem Maschinenwort sensibel ist, werden auch Bits mitberechnet, für die es keine Änderung gab. Die Aktivität in der Schaltung steigt sozusagen und der Geschwindigkeitsvorteil der Ereignissteuerung geht etwas verloren. Dennoch ist auch hier ein signifikanter Vorteil vorhanden.

Bei der parallelen Fehlersimulation wird hauptsächlich in zwei Varianten unterschieden, in die musterparallele und in die fehlerparallele Fehlersimulation.

### 3.2.2.1 Musterparallele Fehlersimulation

Die einfachste Methode der Parallelisierung ist die musterparallele Simulation. Sie wird auch als *parallel pattern single fault simulation* (PPSF) bezeichnet. Sie ist sowohl bei der einfachen Logiksimulation als auch bei der Fehlersimulation möglich. Hier werden Eingangsmuster, die zu verschiedenen Zeitpunkten anliegen, nebenher berechnet. Die Abbildung 3.6 verdeutlicht dies anhand einer Wortbreite von 4 Bit bei einem AND-Gatter.

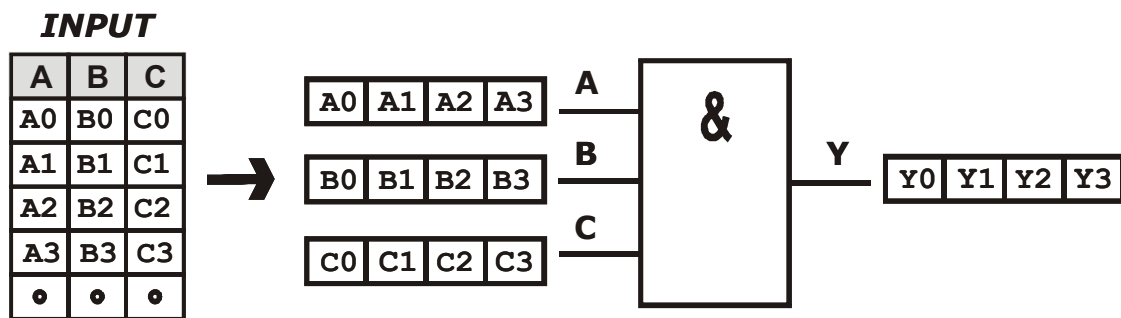


Abbildung 3.6: Musterparallele Berechnung in einem 4-Bit-Maschinenwort

Neben der Parallelisierung bezüglich eines Maschinenwortes lässt sich dieses Verfahren auch auf Vektorrechnern [64] oder durch verteiltes Rechnen auf PC-Clustern anwenden. Bei kombinatorischen Schaltungen kann man das musterparallele Vorgehen besonders gut anwenden, denn hier sind die einzelnen Muster unabhängig voneinander. Bei sequenziellen Schaltungen ist dieser Weg schwieriger, da diese Schaltungen durch den Einsatz von Speichern ein Gedächtnis haben. Ein Folgezustand ist selten unabhängig vom vorhergehenden. Wird also ein Folgemuster zeitgleich errechnet, ohne dass der korrekte Schaltungszustand vorliegt, kann das Ergebnis nicht korrekt sein. Dennoch wurde gezeigt, dass sequenzielle Schaltungen musterparallel simuliert werden können [19,20]. Hier kommen beispielsweise Heuristiken zum Einsatz, die einiges an Rechenaufwand einfordern. So geht der Geschwindigkeitsvorteil des Parallelen zum Teil wieder verloren.



### 3.2.2.2 Fehlerparallele Fehlersimulation

Beim fehlerparallelen Verfahren werden zunächst an den primären Eingängen im gesamten Wort der gleiche Bit-Wert angelegt. Die Parallelisierung entsteht dadurch, dass für jedes Bit, abgesehen vom Referenzbit, in der Schaltung ein anderer Fehler injiziert wird. In der Abbildung 3.7 ist dies für ein AND-Gatter zu sehen.

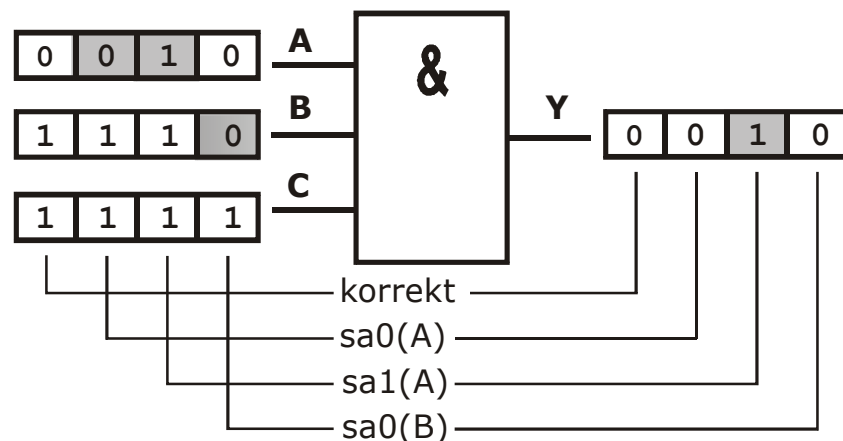


Abbildung 3.7: Fehlerparallele Berechnung in einem 4-Bit-Maschinenwort

Das erste Bit ist hier das Referenzbit und es wird durch die Simulation mit der korrekten Schaltung erzeugt. Die anderen Bits entstehen durch die verschiedenen Fehlerinjektionen. In der Abbildung 3.7 wurden sie durch Änderungen der Leitungswerte erzeugt und durch Graufärbung hervor gehoben. Zum Beispiel bedeutet hier ein sa0(A) einen Haftfehler auf logisch '0' am Eingang A ( $\rightarrow 4.3$ ). Beim eingefärbten Bit in der Ausgabe ergibt sich ein Unterschied, so dass mit einem Vergleich gegen das Referenzbit, welches sich ganz links befindet, dieser Fehler erkannt wird. Durch die Fehlerinjektion und Fehlerfortschaltung kommt es vor, dass einige Bits in einem Maschinenwort sich vom Referenzbit unterscheiden.

### 3.2.3 Deduktive Fehlersimulation

Bei der deduktiven Fehlersimulation werden alle Fehler für ein Muster in einem Durchgang abgearbeitet [16]. Das ist durch eine Methode möglich, die man als Fehlerlisten-Fortpflanzung bezeichnen könnte. Dabei hat jede Signalleitung eine eigene Liste, die Fehler enthält, welche dort einen Fehlereffekt erzeugen. Zusätzlich existieren für alle Komponenten Verknüpfungsregeln, die abhängig von der Eingangsbelegung über eine Fehlerfortpflanzung entscheiden. Bei der deduktiven Fehlersimulation beginnt man an den Eingängen mit einer Belegung und arbeitet sich sukzessiv durch die Schaltung. Am Ende liegt für jeden Ausgang eine fertige Liste vor, die aussagt, welche Fehler dort erkannt wurden. Bei großen Schaltungen kann der Speicherbedarf den die

Listen erfordern sehr groß werden. Da man die Listen dynamisch erst während des Ablaufes erzeugt, kann es in ungünstigen Fällen zu einem Speicherüberlauf kommen. Diese Methode eignet sich auch für asynchrone, sequenzielle Schaltungen [7].

### 3.2.4 Concurrent Fehlersimulation

Bei der Concurrent oder auch nebenläufigen Fehlersimulation werden, wie auch bei der deduktiven Fehlersimulation, alle Fehler für ein Muster in einem Durchlauf ermittelt [17]. Hier werden nur die Abweichungen der fehlerhaften Schaltung von der korrekten Schaltung simuliert. Im Gegensatz zur deduktiven Methode sind die Fehlerlisten nicht den Leitungen sondern den Gattern zugeordnet. Für jeden aktiven Fehler wird eine Gatterkopie erzeugt. In einer Liste stehen die Namen der Fehler und die entsprechenden Signalwerte. Für einen Musterdurchlauf gilt ein Fehler als erkannt, wenn am Ende eine Kopie des Fehlers als Eintrag an mindestens einem Ausgang vorliegt. Ändert sich im nächsten Simulationsschritt das Eingangsmuster, werden nur die Fehlerlisteneinträge aktualisiert, für die sich Änderungen oder Ereignisse ergeben haben. Nur bei einer Berechnung der fehlerfreien Schaltung werden alle Listeneinträge ungültig. Dieses Vorgehen spart gegenüber der deduktiven Methode Rechenzeit. Nachteilig ist der unter Umständen hohe und nicht vorhersehbare Speicherbedarf. Dieser hängt von der Fehleraktivität bei einem gegebenen Muster ab. In [18] wurde gezeigt, dass diese Methode dank ihrer Flexibilität auch auf anderen Ebenen als der Gatterebene eingesetzt werden kann.

Die vorgestellten Fehlersimulationsalgorithmen können als Basisalgorithmen angesehen werden. Neben diesen Basissimulationen gibt es leistungsfähige Simulatoren wie HOPE [21] und PROOFS [22], die gleich mehrere der vorgestellten Varianten innehaben und zur Simulationsbeschleunigung nutzen. Eine Rechnung mittels Boole'scher Differenzen kann bei kombinatorischer Logik auch zur Fehlerdetektion genutzt werden. Es existieren auch Vorschläge für eine symbolische Simulation von Hardwarebeschreibungen [15,14].

Bei den hier vorgestellten Verfahren gibt es allerdings eine deutliche Einschränkung. Sie arbeiten fast ausschließlich mit dem klassischen Haftfehlermodell. Jedoch lassen sich nicht alle physikalischen Defekte mit diesem Fehlermodell aufdecken, weshalb es auch Anstrengungen gibt, nichtklassische Fehler zu simulieren [12,11].

Eigene Untersuchungen zu Verfahren der Simulationsbeschleunigung im Zusammenhang mit dieser Arbeit findet man im Kapitel 7.

### 3.3 Bisherige Fehlersimulationen in SystemC

Die im Kapitel 3.2 vorgestellten Verfahren der Fehlersimulation waren in erster Linie auf das klassische Haftfehlermodell beschränkt und überwiegend für die Gatterebene vorgesehen. Eine Vorgabe bezüglich einer Programmiersprache oder die Art der Schaltungsbeschreibung, wie z. B. auf ein bestimmtes Netzlistenformat, ist dort unnötig. Die Verfahren sind relativ frei in der Art ihrer Implementierung.

Deshalb könnte man prinzipiell auch davon ausgehen, dass eine Fehlersimulation mittels SystemC möglich ist. Schließlich gibt es auch einige Publikationen zu Fehlersimulationen die VHDL-Werkzeuge und -Beschreibungen verwenden [144,140]. Außerdem erweist sich der SystemC-Simulationskernel dem eines konventionellen HDL-Simulators bezüglich der Ausführungsgeschwindigkeit meist als überlegen [123]. Bisher ist aber keine Publikation bekannt, die für eine typische Fehlersimulation in Hardware SystemC verwendet. In [114] wird eine „Error Simulation“ vorgestellt (→5.3.1.1), was jedoch dem Fall einer Fehlersimulation ähnelt. In [38, S.144] wird die Validierungsumgebung mit Fehlerinjektion (→5.3.1.2) auch als *high-level fault simulation* bezeichnet. Um die für eine Fehlersimulation nötige Rechengeschwindigkeit zu erreichen, wird hierzu z. B. das fehlerhafte DUV in Hardware synthetisiert. Dieser Vorgang ist aber laut Definition dann keine Simulation, sondern eine Emulation.



## 4 Hardware- und Fehlermodellierung

Wesentliche Voraussetzungen für eine simulationsbasierte Fehlerinjektion sind Kenntnisse über das Aussehen der Hardware und die Modellierung der zu simulierenden Fehler. Hierbei kann die Komplexität unter Berücksichtigung aller Gesichtspunkte recht groß werden. So lässt sich die zu verwendende Hardware aus verschiedenen Blickwinkeln betrachten. Man kann die Hardware z. B. bezüglich der zu modellierenden Ebene in Struktur, Verhalten und Geometrie untersuchen (→2.1).

Im Folgenden soll der Zusammenhang zwischen der Funktionalität und der Schaltungstechnologie [92] näher betrachtet werden. Die Funktionalität spiegelt sich im Verhalten wider und wird durch entsprechende Komponenten realisiert. Andererseits findet man in einer Technologie nur eine bestimmte Menge an zu modellierenden Komponenten wieder. Für die zu betrachtenden HW-Applikationen ergibt sich daher ein zweidimensionales Feld. Berücksichtigt man noch die Möglichkeiten im Zusammenhang mit den zu verwendenden Fehlermodellen, so entsteht ein dreidimensionaler Raum.

In der Abbildung 4.1 ist das Problem in Form eines Quaders angedeutet. In der Horizontalen sind verschiedene Schaltungstechnologien aufgetragen. In der Vertikalen sieht man verschiedene Funktionalitäten; hier sind es diverse Logik-Realisierungen. In die Tiefe zeigenden Richtung sind einige Fehlermodelle dargestellt. Der Eintrag eines Punktes kennzeichnet, dass für die jeweilige Zelle eine Realisierung möglich ist. Die Abbildung 4.1 erhebt nicht den Anspruch nach Vollständigkeit, sondern verweist auf gebräuchliche Implementierungen. Der Anspruch auf Absolutheit ist schon dadurch nicht gegeben, dass der technologische Fortschritt zu neuen Technologien führt. Dennoch zeichnet sich ab, dass für die Abbildung 4.1 eine vollständige Untersuchung aller Fälle recht umfangreich wird.

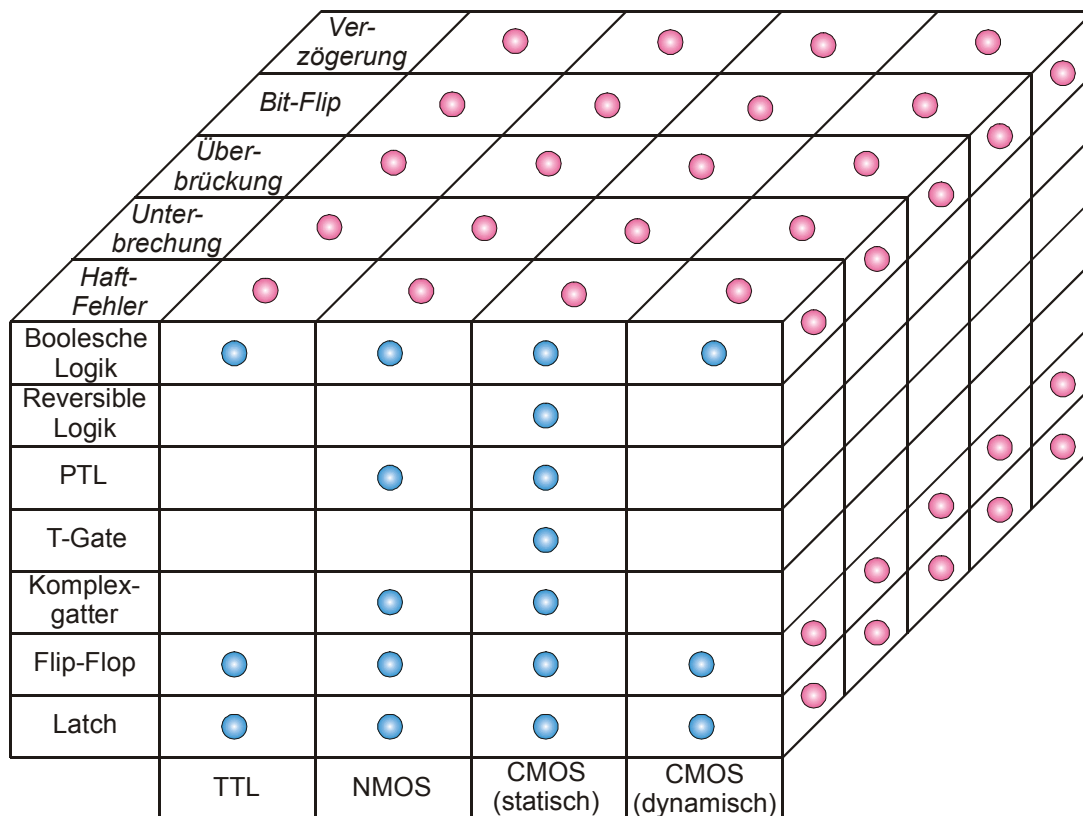


Abbildung 4.1: Zusammenhang zwischen Schaltungstechnologien, Verhalten und Fehlermodellen

Dieses Kapitel ist so aufgebaut, dass zunächst einige Ebenen des HW-Entwurfs mit Beispielen vorgestellt werden. Neben den bisher schon in SystemC modellierbaren RT- und Logikebenen wird eine neue Entwurfsmöglichkeit für die Schalterebene gezeigt. Dem schließen sich diverse Realisierungen von HW-Komponenten an, welche sich nicht in anderen Simulatoren und auch HDL-Programmen wiederfinden. Die Betrachtungen in dieser Arbeit bezüglich der Hardware erstrecken sich auf die Register-Transfer-Ebene und unterhalb dieser. Zwar ist es auch möglich, z. B. Hardware auf der algorithmischen Ebene zu beschreiben, allerdings lässt eine Formulierung auf dieser Stufe noch eine Implementierung in Software zu. Zur Beschreibung eines HW-Systems kommen neben C/C++-Sprachkonstrukten auch Mittel aus der Standard-SystemC-Bibliothek zum Einsatz (→2.2).

Nach der Hardware-Modellierung folgt eine Aufstellung typischer Fehlermodelle. Für diese Fehlermodelle werden für eine erfolgreiche Simulation mit Fehlern Umsetzungen in SystemC vorgestellt.

## 4.1 Hardware-Modellierung in SystemC

### 4.1.1 Die Register-Transfer-Ebene in SystemC

Die Register-Transfer-Ebene oder auch Register-Transfer-Level (RTL) beschreibt die Schaltungseigenschaften in Form von Operationen und von Transfers der zu verarbeitenden Daten. Das zeitliche Verhalten wird durch Takt- und Setz- bzw. Rücksetzsignale definiert. Beim Verhalten wird häufig eine Unterscheidung in Steuerfluss und Datenfluss vorgenommen. Komponenten der RT-Ebene sind überwiegend arithmetisch-logische Einheiten (ALUs), Register, Speicher, Multiplexer und Decoder. Aber auch endliche Zustandsautomaten (FSMs) werden in ihrer Modellierung für gewöhnlich der RT-Ebene zugerechnet.

Eine Einschränkung der zu verwendenden Befehle und Funktionen aus dem C++/SystemC-Sprachschatz ist allgemein nicht festgelegt. Jedoch lassen sich RTL-Beschreibungen in SystemC direkt in Hardware synthetisieren [182,179]. Für diese Fälle existieren Beschränkungen, weshalb man hierzu nur eine festgelegte Untermenge des Sprachschatzes verwenden darf [180,181]. Um einen minimalen Grad an Kompatibilität zwischen den verschiedenen Sprachen und Werkzeugen zu bekommen, wurden z. B. in [193] Vorschläge unterbreitet. Für die reine Simulation sind diese Einschränkungen unerheblich. Bezüglich der zu verwendenden Datentypen sind sowohl C++- als auch SystemC-Typen zugelassen.

Die Möglichkeit des Entwurfs auf der RT-Ebene wurde schon bei der ersten SystemC-Version berücksichtigt und ist in unterschiedlichen Literaturquellen ausgiebig beschrieben [39,37,35,182]. Deshalb wird hier nur exemplarisch eine kleine FSM vorgestellt. Die Abbildung 4.2 zeigt die FSM einer einfachen Temperaturregelung.

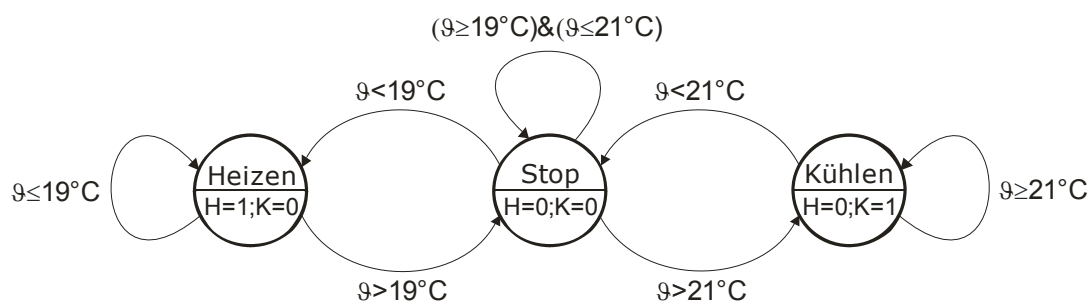


Abbildung 4.2: FSM einer einfachen Temperaturregelung

Die Steuerung besitzt drei Zustände:

- einen Stop-Zustand,
- einen Heizen-Zustand und
- einen Kühlen-Zustand.

Abhängig von einer gemessenen Temperatur 9 erfolgt ein Zustandswechsel oder es wird im aktuellen Zustand verblieben und eine Ausgabe an den Aktoren *H* und *K* getätigt.

Das Listing 4.1 zeigt einen Ausschnitt aus der FSM-Realisierung in SystemC. Die drei Zustände sind in Zeile 7 in einem Auszählungstyp *state\_t* implementiert und diesem Typ werden zwei Signalen zugewiesen. Das eine Signal kennzeichnet den aktuellen Zustand *state* und das andere Signal den bei der nächsten positiven Taktflanke von *clk* einzunehmenden Zustand *next\_state*. Die Auswertung der Eingaben erfolgt in der Methode *op\_logic()*. In dieser Methode werden in Abhängigkeit vom Zustand die Aktoren angesteuert und in Abhängigkeit der Eingabe der nächste Zustand ermittelt, aber noch nicht aktiviert.

---

```
2  SC_MODULE(rtl_fsm) {
    sc_in_clk clk;
4   sc_in<int> temp;           //Temperaturmessung
    sc_out<bool> H, K;         //Ansteuerung der Aktoren
6
    enum state_t { stop, heizen, kuehlen };
8   sc_signal<state_t> state, next_state;

10  void op_logic();           //Methode der Verarbeitungslogik
    void update_state();       //Übergang zum nächsten Zustand
12
    SC_CTOR(ex_fsm) {
14        SC_METHOD(update_state);
        sensitive_pos << clk;
15        SC_METHOD(op_logic);
        sensitive << state << temp;
16        state = stop;
    }
    . . .
18 . . .
    void ex_fsm::update_state() { state = next_state;}
20
    void ex_fsm::op_logic() {
21        switch(state) {
            case stop:
22            H.write(0);
            K.write(0);
24            if (temp.read() < 19) next_state = heizen;
            else
26                if (temp.read() > 21) next_state = kuehlen;
                else next_state = stop;
28            break;
        }
    }
    . . .

```

---

Listing 4.1: FSM einer Temperaturregelung in SystemC



### 4.1.2 Die Logik-Ebene in SystemC

Die Logik- oder auch Gatterebene beschreibt das Systemverhalten mittels logischer Verknüpfungen. Die Betrachtung des Timing-Verhaltens geht zum Teil so weit, dass man mitunter auch Verzögerungszeiten von Gattern berücksichtigt. Dies ist aber nicht zwingend. Komponenten dieser Entwurfsebene sind z. B. AND- und OR-Gatter und auch Flip-Flops. Bezüglich der zu verwendenden Datentypen finden die Logik-Typen der SystemC-Bibliothek Verwendung. Aus den im Abschnitt 2.3 erwähnten Gründen, ist die Modellierung auf dieser Ebene selten anzutreffen. Komponenten dieser Logikebene finden sich aber oft in RTL-Beschreibungen wieder.

Eine Gatter-Beschreibung wird manchmal auch als boolesches Funktionsmodell bezeichnet und in der Abbildung 4.3 ist sie als AND-Komponente dargestellt.

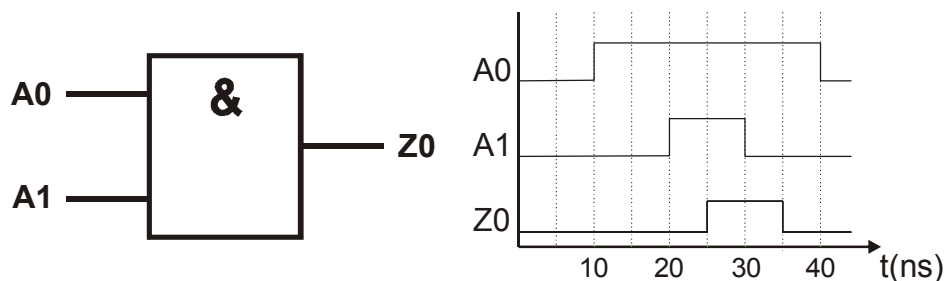


Abbildung 4.3: AND-Modul mit 5 ns Verzögerung

---

```
2  SC_MODULE (and2) {  
    sc_out<sc_logic> z0;  
4    sc_in<sc_logic> a0, a1;  
  
6    void and2calc();  
  
8    SC_CTOR (and2) {  
        SC_METHOD (and2calc);  
10       sensitive << a0 << a1;  
        . . .  
  
12 void and2::and2calc() {  
    next_trigger(5, SC_NS);  
14    z0.write (a0.read() & a1.read());  
    . . .  
}
```

---

Listing 4.2: AND-Modul mit Verzögerung in SystemC

Das Listing 4.2 zeigt einen Teil aus der Umsetzung des AND-Moduls in SystemC. In der Zeile 13 erfolgt die Angabe der Verzögerung.

Mit dem `next_trigger()`-Befehl wird eine dynamische Sensitivität mit Zeitverzögerung erreicht, die für die Ausgabe zuständig ist. Die Reaktion auf eine Eingangsänderung bestimmt sich durch die Sensitivitätsliste in Zeile 10. Die Ausgabeoperation der AND-Funktion findet in Zeile 14 statt.

### 4.1.3 Die Schalterebene in SystemC

Die Schalterebene (engl. *switch-level*) ist eine Zwischenstufe bei der Verfeinerung des Entwurfs von der Logikebene zur elektrischen Ebene. Zum einen bietet die Modellierung auf dieser Stufe eine höhere Genauigkeit als die Logikebene und zum anderen ist die Simulationszeit deutlich kürzer als die Berechnung von kontinuierlichen Werten auf Basis von Differentialgleichungen auf der elektrischen Ebene.

Die Modellierung der Transistoren erfolgt in Form von Schaltern. Dies ist typisch für den VLSI-Entwurf. Das bidirektionale Verhalten der Simulation ist die wesentliche Eigenschaft auf der Schalterebene. Weiterhin ist manchmal die Berücksichtigung von Widerständen  $R$  und Kapazitäten  $C$  zu finden. Mit diesen Elementen lassen sich z. B. zusätzliche Zeiteigenschaften, wie sie durch das Speichern von Ladungen in Kapazitäten hervorgerufen werden, modellieren.

Lange Zeit wurde die Schaltersimulation nicht als eigenständige Ebene angesehen (→2.1), da die früher eingesetzten bipolaren Transistoren wenig mit einem idealen Schalter gemein hatten. Mit der starken Verbreitung der MOS-Technik und deren geringen Rückwirkung der Kanäle auf die Gates bekam die eigenständige Betrachtung der Schalterebene Bedeutung [1]. Man schätzt zum Beispiel, dass im Jahr 2002 etwa 90 % aller gefertigten integrierten Schaltungen in CMOS-Technologie erstellt wurden [101].

Die Modellbetrachtung auf der Schalterebene bietet eine Reihe von Vorteilen. So kann es in Schaltungen nötig sein, dass zum verbesserten Treiben eines höheren Stromes mehrere Transistoren parallel betrieben werden. Auf der Logikebene würde aber nur ein Single-Buffer erscheinen [95]. Die Betrachtung, insbesondere beim Ausfall eines Treibers, wäre hier zu ungenau. Der Nutzen in der Schaltermodellierung liegt also in der höheren Genauigkeit. Noch deutlicher wird der Vorteil der Schalter-Simulation in Kombination mit der konkreten Simulation von Fehlern. So besitzt z. B. ein einfaches NAND-Gatter in CMOS-Technologie auf der Logikebene drei Knoten, nämlich die beiden Eingänge und den Ausgang. Bei der Untersuchung des Schalter-Modells ergibt sich ein zusätzlicher interner Knoten, der bei der Logikbetrachtung nach außen nicht sichtbar wird [91]. Tritt nun für diesen Knoten ein Brückenfehler auf, so kann es sein, dass dieser von einem Testmuster nicht entdeckt werden kann, da das Testmuster-erzeugende ATPG-Werkzeug auf der Logikebene arbeitete.

Für SystemC ist bisher keine Modellierung auf der Schalterebene bekannt (→2.3). Dieser Umstand ist nicht nur aus den eben genannten Vorteilen der Schaltersimulation ein Mangel, sondern stört auch den durchgehenden Designablauf (→2.4) in einer

Entwurfsumgebung. Während die RTL- und Logikebene mit den SystemC-Klassen gut zu modellieren ist ( $\rightarrow$ 4.1.1, 4.1.2) und auf der analogen elektrischen Ebene das SystemC-AMS zunehmende Akzeptanz erfährt [48], blieb die Schalterebene bisher unbeachtet.

Im Folgenden wird die Modellierung der Schalterebene in SystemC vorgestellt. Die Modellierung der Einflüsse von expliziten Kapazitäten und insbesondere von Widerständen blieb hierbei zunächst unbeachtet. Jedoch lassen sich zeitliche Eigenschaften implementieren, wie im Abschnitt 4.1.2 und im Abschnitt 4.3.2.4 vorgestellt. Zwei Hauptaufgaben wurden bei der Umsetzung der Schalter-Modellierung gelöst. Zum einen wurde der Logik-Datentyp modifiziert und zum anderen wurden die Schalter realisiert.

Um ein genaueres Hardware-Verhalten zu erreichen, wurde die Logikimplementierung in SystemC auf den Umfang der in der IEEE 1164 definierten 9-wertigen Logik erweitert [30]. Die hierzu notwendigen Änderungen sind in der folgenden Übersicht aufgeführt:

1. Der Datentyp *sc\_logic* wurde um die zusätzlichen Werte erweitert.
2. Einige Methoden und Operatoren mussten für ein sinnvollen Umgang umgeschrieben werden.
3. Für die Typumwandlung wurde die Konvertierungstabelle modifiziert.
4. Die neuen Signalwerte bekamen in der Klasse *sc\_signal\_resolved* Signalstärken zugewiesen.
5. Weitere Methoden, z. B. zur Unterstützung der Simulation, wurden entsprechend angepasst.

Mit der Einführung der IEEE-1164-Konformität erweiterte sich der Wertebereich der Klasse *sc\_logic*. Der neue Wertebereich lautet somit:

$$sc\_logic \in \{0, 1, Z, X, L, H, U, W, D\}$$

bzw. nach SystemC-Syntax:

$$sc\_logic \in \{sc\_logic\_0, sc\_logic\_1, sc\_logic\_Z, sc\_logic\_X, sc\_logic\_L, sc\_logic\_H, sc\_logic\_U, sc\_logic\_W, sc\_logic\_D\}$$

Die Signalstärken wurden auch entsprechend des Standards IEEE 1164 in der Klasse *sc\_signal\_resolved* umgesetzt. Damit ergaben sich für die neuen Werte die folgende Bedeutungen:

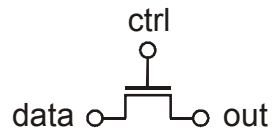
- *sc\_logic\_L* - bedeutet eine schwache '0' und dient z. B. der Modellierung eines Signals mit einem Pull-up-Widerstand gegen Masse;
- *sc\_logic\_H* - bedeutet eine schwache '1' und dient z. B. der Modellierung eines Signals mit einem Pull-up-Widerstand gegen die Betriebsspannung;

- *sc\_logic\_U* - kennzeichnet ein nicht initialisiertes Signal, z. B. betrifft dies oft Speicherelemente während der Elaboration oder beim Simulationsstart;
- *sc\_logic\_W* - bedeutet ein schwaches unbekanntes Signal, dies tritt z. B. bei einem Zusammentreffen von *sc\_logic\_L* und *sc\_logic\_H* auf;
- *sc\_logic\_D* - bedeutet den Zustand egal (don't care), dieser Zustand wird häufig in der Logikoptimierung verwendet. Im Standard IEEE 1164 ist dieser Wert mit '-' abgebildet, damit es in den SystemC-Klassen nicht zu Unstimmigkeiten mit dem Operator '-' kommt, wurde von der Bezeichnung „don't care“ das *D* als Symbol übernommen, welches z. B. auch zum *sc\_logic\_D* führte.

Die bisher schon vorhandenen Werte *sc\_logic\_0*, *sc\_logic\_1*, *sc\_logic\_X* bekamen die Bedeutung von starken Werten, abgesehen vom Tristate-Zustand *sc\_logic\_Z*. Um die neuen Werte sinnvoll verwenden zu können, mussten die logischen Operatoren &, |, ^, ~ erweitert werden. Für die davon abgeleiteten Operatoren wie z. B. |= ergab sich die Anpassung automatisch. Die Operator-Tabellen und die Auflösungstabelle sind so organisiert, wie im Standard IEEE 1164 festgelegt. Die genaue Umsetzung kann dort entnommen werden [30].

Die Hauptkomponenten auf der Schalterebene beschränken sich auf die Schalter in Form von Transistoren. Da die Schalterebene vorzugsweise (siehe oben) in der CMOS-Technologie von Bedeutung ist, wurden die hierfür relevanten NMOS- und PMOS-Transistoren in SystemC implementiert. Während in der Hardware-Beschreibungssprache VHDL keine standardmäßige Modellierung auf der Schalterebene vorgesehen ist, lässt sich in Verilog diese Ebene abbilden. In Verilog existieren hierzu sogenannte *MOS Switches* als *Primitives*, welche im Standard IEEE 1364 definiert sind [96]. Der NMOS-Transistor leitet bei einer höheren positiven Spannung am *Gate*-Anschluss und der PMOS-Transistor bei wenig Spannung am Steuereingang.

Die Abbildung 4.4 zeigt die Wahrheitstabelle und das Symbol für den NMOS-Transistor in Anlehnung an [96] ausführlich. In der Abbildung 4.4 schaltet der Transistor bei einem Werte von '1' am Kontrolleingang *ctrl*, welcher der Gate-Pin ist, einen Wert vom Eingang *data* zum Ausgang *out*. Die Durchschaltung erfolgt dabei verlustfrei, d. h., der Eingangswert erscheint auch am Ausgang. Eine '0' am Kontrolleingang bedeutet einen offenen Schalter, dies wird durch den hohen Impedanzwert 'Z' symbolisiert. Die Tabelle zeigt für die Werte 'X' und 'Z' die neu eingeführten Werte 'L' und 'H'. Der Hintergrund dieser Festlegung ist weniger trivial als bei der '0' bzw. '1'. Er beruht auf das physikalische Prinzip, dass ein Ausgangsknoten eine Ladung speichern kann. Somit bezieht sich der entstandene Wert auf einen früheren Zustand [97]. Für den PMOS-Transistor sieht die Wahrheitstabelle ähnlich aus, in dieser sind nur die Inhalte der Spalten '0' und '1' für *ctrl* miteinander vertauscht (→Anhang A.1.2).



NMOS	ctrl			
data	0	1	X	Z
0	Z	0	L	L
1	Z	1	H	H
X	Z	X	X	X
Z	Z	Z	Z	Z

Abbildung 4.4: NMOS-Primitive der Schalterebene

Das Listing 4.3 zeigt den NMOS-Schalter in SystemC in Anlehnung an die IEEE 1364.

```

2  SC_MODULE (nmos) {
    sc_out<sc_logic> z0;
4   sc_in<sc_logic> a0, ctrl;
    sc_logic a0_l, ctrl_l, res_l;
6   int a0_t, ctrl_t, res_t;

8   void doit();

10  SC_CTOR (nmos) {
    SC_METHOD (doit);
12    sensitive << a0 << ctrl;
    z0.initialize(sc_logic_Z);
14  ...

16  const char nmosTab[4][4] = {
    //      0      1      Z      X
18      { 'Z', '0', 'L', 'L' },    //0
    { 'Z', '1', 'H', 'H' },    //1
20      { 'Z', 'Z', 'Z', 'Z' },    //Z
    { 'Z', 'X', 'X', 'X' },    //X
22  };

24  void nmos::doit() {
    a0_l = a0.read();
26    a0_t = (int) a0_l.value() ;
    ctrl_l = ctrl.read();
28    ctrl_t = (int)ctrl_l.value();
    z0.write((sc_logic)nmosTab[a0_t] [ctrl_t]);
30 }

```

Listing 4.3: Einfacher NMOS-Transistor in SystemC nach IEEE 1364

Die Funktionalität ist hier, wie auch in der Abbildung 4.4 zu sehen, mit Hilfe einer Tabelle implementiert (Zeilen 16-22). Das gewährleistet bezüglich des Programmcodes konstante und damit deterministische Ausführungszeiten. Zudem ist die Tabellenzugriffszeit kürzer als die mittlere Zugriffszeit bei der Funktionsimplementierung mit bedingten Anweisungen, da deutlich weniger Befehle

ausgeführt werden müssen. Für die Zuweisung des korrekten Wertes sind einige Zwischenschritte mit zusätzlichen Variablen nötig, da z. B. eine direkte Typumwandlung bzw. Zuweisung nicht möglich ist (Zeile 26). Der Ausgabewert an *z0* wird durch die beiden Eingangswerte an *a0* und *ctrl* aus der Tabelle *nmosTab* bestimmt.

In Verilog existieren *Primitives* in der Form bidirektionaler *pass-switches*. Kennzeichnend hierfür ist, dass die Ein- und Ausgänge bidirektionaler Natur sind – d. h., ein Anschluss ist sowohl als Eingang wie auch als Ausgang realisiert. In SystemC wird dies durch ein Ersetzen zu *sc\_inout* in der Portdeklaration erreicht (Zeilen 3, 4)

Ein Nachteil ist in der Implementierung nach der IEEE 1364 in der Abbildung 4.4 vorhanden. Zur Logik-Darstellung wird ein 9-wertiger Typ verwendet, in der Wahrheitstabelle sind aber in der Spalte und Zeile nur die vier Werte '0', '1', 'Z', 'X' berücksichtigt. Bei dem Modell nach Listing 4.3 und mit einer Eingabe eines Wertes, für den keine Ausgabe festgelegt ist, erfolgt ein Zugriff auf eine Speicherzelle, die u. U. außerhalb des Tabellenbereichs liegt. Der Rückgabewert ist somit inkorrekt. Eine vollständige Implementierung ist in der Tabelle 4.1 zu sehen.

NMOS	ctrl								
data	0	1	Z	X	L	H	U	W	D
0	Z	0	L	L	Z	0	X	L	X
1	Z	1	H	H	Z	1	W	H	W
Z	Z	Z	Z	Z	Z	Z	W	Z	W
X	Z	X	X	X	X	X	X	X	X
L	Z	L	L	L	Z	L	W	L	W
H	Z	H	H	H	Z	H	W	H	W
U	U	U	X	X	U	U	X	X	X
W	Z	W	W	W	Z	W	W	W	W
D	Z	D	W	W	Z	D	W	W	W

Tabelle 4.1: Erweitertes *NMOS-Primitive* der Schalterebene

In der Darstellung in der Tabelle 4.1 erfolgt die Durchschaltung verlustlos, d. h., ein Eingangswert erscheint in Durchflussrichtung auch am Ausgang. Die Umsetzung in SystemC erfolgt äquivalent zum Listing 4.3. Die Organisation der Funktion in Form einer Tabelle hat auch den Vorteil, dass das IEEE-1364-Modell und das hier vorgestellte erweiterte Modell gleiche Ausführungszeiten haben.

In einigen Anwendungen gilt es zu beachten, dass der NMOS-Transistor die '0' gut leitet und die '1' nur schwach überträgt. In der Tabelle 4.1 müsste sich für diesen Fall z. B. bei *ctrl* = 1 und *data* = 1 statt einer '1' ein 'H' wieder finden. In [98] wurde eine solche verlustbehaftete Variante für eine 9-wertige Logik vorgestellt. Eine nach [98] ähnliche Implementierung für den NMOS-Transistor befindet sich im Anhang A.1.3.

Für Verilog existieren hierzu widerstandsbehaftete *Primitives* (*rpmos*, *rnmos*), die eine Reduzierung der Signalstärke vornehmen. In [99] wird sogar eine sehr genaue Schalter-Modellierung vorgestellt, die mit 256 Zwischenwerten arbeitet. Welches Schaltermodell zu wählen ist, hängt vom jeweiligen Anwendungsfall und von den Eigenschaften der gegebenen Bauelemente ab. Bei langen seriellen Zusammenschaltungen dürfte die verlustbehaftete Übertragung der Source-Drain-Strecke nicht unberücksichtigt bleiben.

## 4.2 Ausgewählte Hardware-Komponenten

Die Modellierung von den gebräuchlichsten Hardware-Komponenten des Schaltungsentwurfs in SystemC wurde schon in den ersten Versionen der Spracherweiterung erläutert [39]. Hierbei lag der Fokus auf Flip-Flops bzw. booleschen Funktionen und deren Zusammenschaltungen, z. B. zu Schieberegistern und Zählern. Außerdem ist die einfache Modellierung auf der Gatter- bzw. Logikebene Bestandteil vieler SystemC-Tutorials [61]. Eine recht umfangreiche Beschreibung der Systemmodellierung mit Hardware-nahen Komponenten findet man in [37]. Neben den weit verbreiteten und bekannten Logikelementen wie AND, OR, etc. und Speicherelementen wie Flip-Flops gibt es in derzeit gefertigten Schaltungen noch weitere Elemente. Solche Elemente sind z. B. Transfer-Gatter oder komplexere Logikfunktionen (→Abb. 4.1).

Im folgenden Abschnitt werden eigene Modellierungen von HW-Komponenten in SystemC vorgestellt, deren Umsetzung auch in anderen HDLs zum Teil selten zu finden ist. Dabei wurde auch das Auftreten von Fehlererscheinungen berücksichtigt.

### 4.2.1 CMOS-Komplexgatter

Diverse logische Funktionen lassen sich in der CMOS-Technologie effizient durch besondere Gatter verwirklichen. Diese Gatter werden als Komplexgatter (engl. *complex gate*) bezeichnet, weil sie komplexe logische Funktionen in einem einfachen Aufbau realisieren. Der Vorteil liegt dabei nicht nur in einem geringeren Platzbedarf, sondern u. U. auch in einer kürzeren Ausführungszeit als bei einer herkömmlichen Logik-Implementierung. Im Folgenden wird gezeigt, welche Probleme bei der Modellierung eines Komplexgatters auftreten können, und es werden dazu Lösungsmöglichkeiten in SystemC vorgestellt.

Die Abbildung 4.5-a zeigt ein AOI32-Komplexgatter mit MOS-Transistoren vom Anreicherungstyp. Die Funktion hierzu lautet:

$$Y = \overline{(A \wedge B) \vee (C \wedge D \wedge E)}$$

Das Ergebnis ist also die Negation der OR-Funktion zweier AND-Verknüpfungen. Die eine AND-Funktion besteht aus zwei und die andere aus drei Signalen. Daraus ergibt sich die Bezeichnung AOI32: AND-OR-INVERT-3,2.

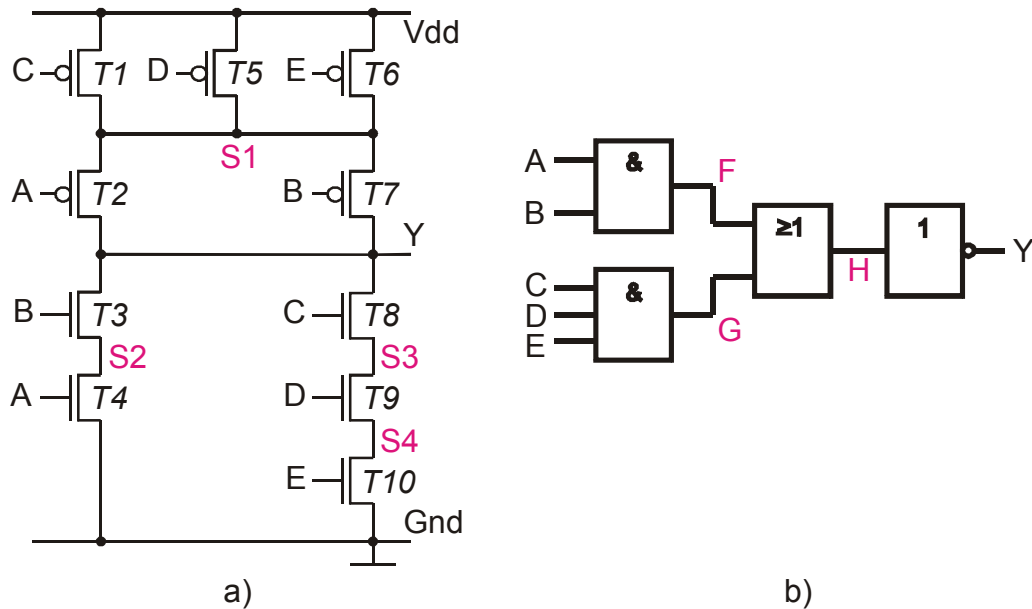


Abbildung 4.5: a) AOI32-Komplexgatter mit b) Logikgatterauflösung

Erfolgt nun beispielsweise eine funktionale Simulation in einem herkömmlichen Simulator, so kann das AOI32-Gatter durch normale Logikgatter in ihm nachgebildet werden. Dies ist in der Abbildung 4.5-b dargestellt. Bei den meisten Logiksimulatoren, die keine Bibliothek von Komplexgattern besitzen, kann so ohne Probleme auch ein Komplexgatter simuliert werden. Bei Fehlersimulationen ist diese Modellierung ungenügend. Viele Fehlersimulatoren beherrschen nur boolesche Grundfunktionen und einige wenige Speicherelemente [22,21]. Erfolgt bei einem solchen Fehlersimulator eine Logikgatterauflösung, so entstehen auf der Gatterebene zusätzliche Schaltungsknoten, die real nicht existieren. Dies ist in der Abbildung 4.5-b zu sehen. Es kommt zu fehlerhaften Simulationsergebnissen, da von den Knoten *F*, *G*, *H* falsche Fehlerberechnungen entstehen. Beherrscht ein Fehlersimulator hierarchische Beschreibungen, so lassen sich die Komplexgatter als „Makrogatter“ definieren, die dann im Stück richtig simuliert werden. Allerdings muss die Fehlerlistengenerierung in der Lage sein, Restriktionen (*constraints*) zu beherrschen, damit es bei der Hierarchieauflösung nicht zur Zerlegung aller Komponenten kommt.

Das Listing 4.4 zeigt den Code für das AOI32-Gatter in SystemC. Die logische Funktion steht in einer einzigen Zeile (Zeile 14). Der gezeigte Ansatz lässt sich an Stelle von SystemC auch mit anderen HDLs wie VHDL oder Verilog lösen.



---

```

2  SC_MODULE (aoi32){
    sc_out<sc_logic> Y;
4    sc_in<sc_logic> A, B, C, D, E;

6    void doit();

8    SC_CTOR (aoi32) {
        SC_METHOD (doit);
10       sensitive << A << B << C << D << E;
        . . .
12 . . .
    void aoi32::doit() {
14     Y.write(!((A.read() & B.read()) | (C.read() & D.read() & E.read())))
        . . .

```

---

Listing 4.4: AOI32-Modul in SystemC

Bereits vor langer Zeit wurde erkannt, dass eine Simulation von diversen Fehlermodellen auf der Gatterebene für Komplexgatter nicht zufriedenstellende Ergebnisse liefert [100]. Interne Schaltungsknoten müssen auch berücksichtigt werden. In der Abbildung 4.5-a sind dies z. B. die Knoten *S1* bis *S4*. Für die Simulation in einer HDL lässt sich das verfeinerte Fehlerverhalten beispielsweise durch zusätzliche Anweisungen im Prozess modellieren. In [140] wurde eine solche funktionale Fehlersimulation für VHDL-Modelle vorgestellt und u.a. ergab sich mit Hilfe von *fault collapsing* ein deutlicher Speed-up ( $\rightarrow 7.1$ ) für diese funktionalen Modelle in der Ausführung. *Fault collapsing* bezeichnet, vereinfacht ausgedrückt, das Streichen von zu simulierenden Fehlern, da durch Abhängigkeiten diese schon in ihren Auswirkungen von anderen Fehlern berechnet bzw. abgedeckt wurden. Die in [140] verwendeten Basismodelle waren jedoch sehr klein. Für das AOI32-Gatter sieht das funktionale Modell umfangreicher aus. Für eine vollständige Fehlerinterpretation müsste man hierfür alle Eingangsbelegungen, alle Fehlermodelle, alle Ein- und Ausgänge und alle Schaltungsknoten berücksichtigen. In der expliziten Form ergeben sich somit 620 Möglichkeiten allein für das klassische Haftfehlermodell ( $\rightarrow 4.3$ ), die zu keinem effizienten Simulationsmodell führen:

$$2^5 \text{ Eingangsbelegungen} \cdot (6 \text{ Ein- / Ausgänge} + 4 \text{ Knoten}) \cdot 2 = 620 \text{ Möglichkeiten}$$

Diese 620 Varianten durch eine Liste von Anweisungen zu berücksichtigen, scheint nicht sehr effektiv. Natürlich lässt sich die Anzahl der Bedingungen noch minimieren, aber trotz Reduzierung bleibt der Aufwand zur vollständigen Verhaltensmodellierung recht umfangreich.

Eine Alternative stellt die Modellierung mit der im Abschnitt 4.1.3 vorgestellten Schalterebene in SystemC dar. In diesem Fall muss keine ausführliche funktionale

Fehlercharakterisierung erfolgen. Das erforderliche Verhalten ergibt sich implizit durch die verwendeten Transistormodelle, den Eingangswerten und dem Fehlermodell.

In [100] wurde auch gezeigt, dass für viele Fehler das Modell der Schalterebene eine ausreichende Genauigkeit bei deutlich schnellerer Ausführung bietet als die Simulation auf der elektrischen Ebene mit einem SPICE-Simulator [141]. Die Abbildung 4.6 zeigt das Schema des AOI32-Modells in SystemC.

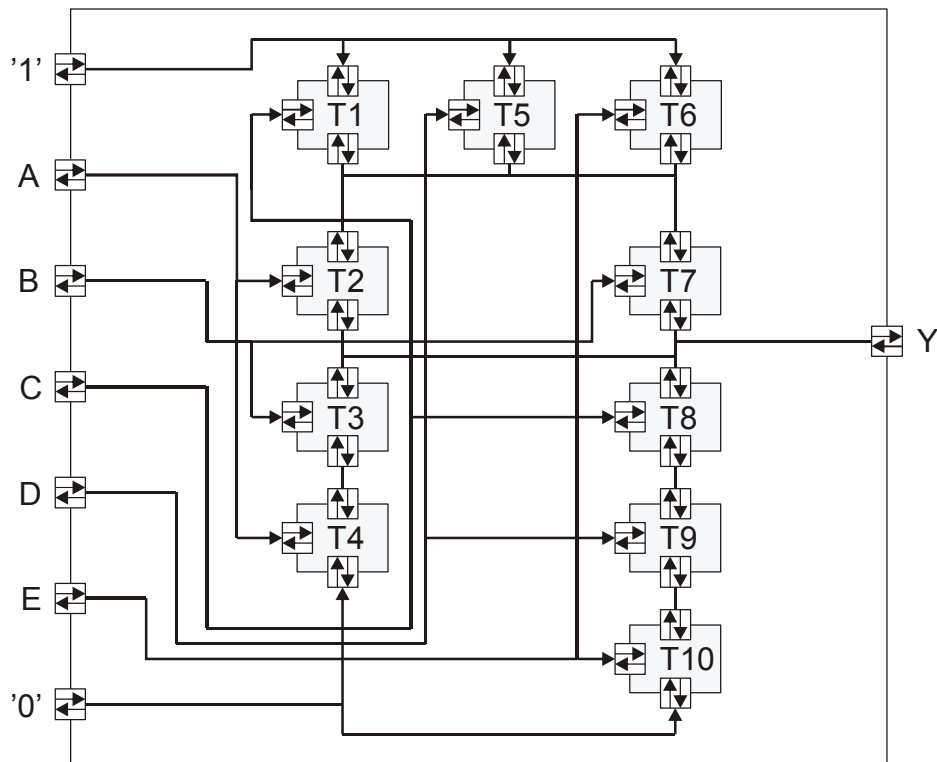


Abbildung 4.6: Struktur des AOI32-Gatters in SystemC

### 4.2.2 Pass-Transistor-Logik

Eine verbreitete Form der Logikimplementierung in NMOS-Schaltungen ist die Pass-Transistor-Logik (PTL) [102]. Daneben gibt es auch kombinierte Varianten aus NMOS- und PMOS-Technologie, wie z. B. die Double-Pass-Transistor-Logic (DPL) [103]. In vielen Bereichen ist die statische CMOS-Logik ( $\rightarrow$  Abb. 4.3,  $\rightarrow$  4.2.1) heutzutage in Gebrauch. Schaltungen in dieser Technologie sind robust und haben hervorragende Rauschtoleranzen. Jedoch verlangen moderne VLSI-Designs höhere Geschwindigkeiten bei geringem Leistungsverbrauch. Eine Alternative sind Schaltungen in dynamischer CMOS-Logik [91,104], welche deutlich schneller sind. Allerdings haben diese dynamischen Schaltungen den Nachteil, dass ihr Rauschverhalten ungünstig ist, was zum Beispiel den Schaltungstest erschwert. Die Pass-Transistor-Logik kann zur statischen CMOS-Logik eine Alternative sein. Sie bietet schnellere Ausführungszeiten bei einer reduzierten Anzahl an Transistoren [105]. Das Hauptproblem bei der PTL bleibt allerdings, dass die Umschaltzeiten auf High länger dauern als die auf Low. Häufig werden zusätzliche Komponenten (Inverter, Repeater) eingesetzt, um dieses Problem zu kompensieren.

Exemplarisch wird in diesem Abschnitt auf die Simulationsmodellierung für ein CPL-Gatter (Complementary-Pass-Transistor-Logic) mit einem Fehlerfall eingegangen. Die Abbildung 4.7 zeigt die Schaltung und das Symbol eines zweifachen CPL-AND/NAND-Gatters. Für den Fall einer Leitungsunterbrechung (stuck-open-Fehler) am Gate-Anschluss des Transistors  $T1$  soll das Verhalten näher untersucht werden.

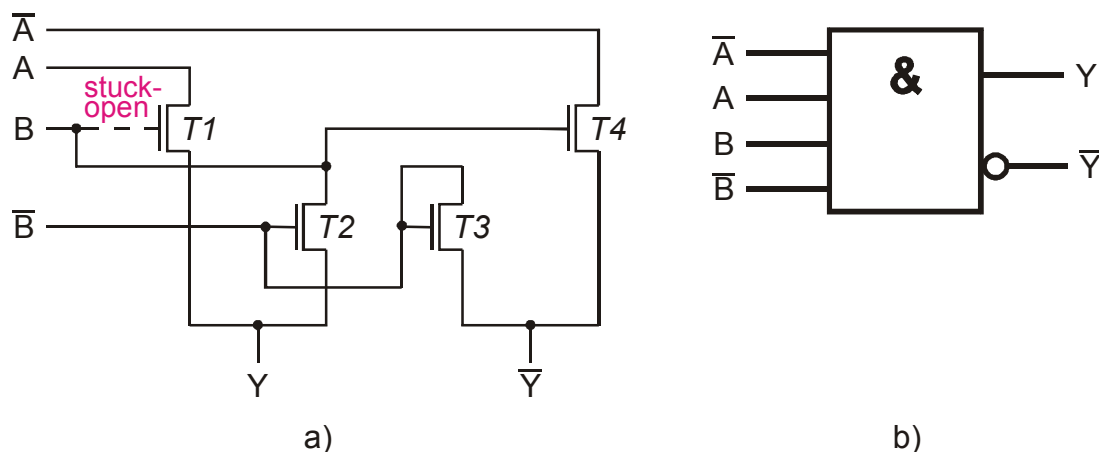


Abbildung 4.7: a) Schaltung und b) Symbol eines CPL-AND/NAND-Gatters

Der komplementäre Ausgang  $\bar{Y}$  arbeitet für den angenommenen Fehlerfall korrekt und eine einfache Fehlermodellierung würde für den  $Y$ -Ausgang ein 'X' liefern ( $\rightarrow$  Tab. 4.2, Spalte 5). Bei genauerer Modellierung kann man die entstehende '0' bei einer '0' am Eingang  $B$  berücksichtigen. Diese Modellierung spiegelt allerdings nicht das wahre

Verhalten wider. Eine Unterbrechung bewirkt in CMOS- und auch in MOS-Schaltungen ein zusätzliches sequenzielles Verhalten [92], welches durch eine Ladungsspeicherung hervorgerufen wird. Im Fall von  $/B = 1$  müsste der  $Y$ -Ausgang den vorherigen Zustand speichernd bewahren. Da der vorherige Zustand aber durch den defekten Transistor  $T1$  nie eine '1' gewesen sein kann, führt der Ausgang eine speichernde '0' ( $\rightarrow$  Tab. 4.2, Spalte 6). Diese Nachbildung ist besser, aber immer noch nicht vollständig.

Im einfachen Fall, dass am Ausgang  $Y$  nur eine unidirektionale, ungetriebene Leitung mit einer anderen Komponente verbunden ist, ergeben sich keine Probleme bei der Modellierung. Zunehmend sind aber Schaltungsteile mit Bussen miteinander verbunden und an diesen Bussen befinden sich oft mehr als zwei Komponenten. Damit es auf den Verbindungen zu keinen Kollisionen kommt, dürfen die angeschlossenen Elemente nur ein treibendes Signal abgeben, wenn sie dazu berechtigt sind. Auf die entsprechende Darstellung eines zwischengeschalteten Tristate-Treibers wurde hier verzichtet. Diese Busse führen im inaktiven Zustand ein hochohmiges 'Z', welches oftmals über Widerstände (Pull-up) zur Betriebsspannung führt. In der Simulation würde sich die modellierte '0' gegen das 'Z' durchsetzen. In Wirklichkeit dominiert in diesem Fall aber das 'Z', da die „parasitäre“ Kapazität des  $Y$ -Ausgangs schnell über die Widerstände zur Betriebsspannung aufgeladen wird ( $\rightarrow$  Tab. 4.2, Spalte 7). In der Tabelle 4.2 sind die einzelnen Sachverhalte noch einmal veranschaulicht.

A	B	Y	/Y	Y(1)	Y(2)	Y(3)	
0	0	0	1	X(0)	0	0	1) vereinfachte Fehlermodellierung 2) an einer Signalleitung 3) an einem Bus
0	1	0	1	X	0	Z	
1	0	0	1	X(0)	0	0	
1	1	1	0	X	0	Z	

Tabelle 4.2: Fehlerverhalten des CPL-AND/NAND-Gatters mit Unterbrechung

Bei der Injektion eines Unterbrechungsfehlers in das Gattermodell wird vor dem Schreiben zunächst geprüft, welcher Logikzustand am Ausgang anliegt und abhängig von diesem entschieden. Da die Klasse `sc_out` von der Klasse `sc_inout` abstammt, kann bei Gebrauch auf dieser auch lesend zugegriffen werden. Bei Bedarf, z. B. zum Zwecke einer besseren Lesbarkeit, empfiehlt es sich, den bidirektionalen Charakter auch durch eine Änderung zu `sc_inout` hervorzuheben. Bei einem 'Z' am Ausgang bleibt der Zustand erhalten und andernfalls wird eine '0' geschrieben. In der Abbildung 4.8 ist der Ablauf in Form eines Pseudo-Codes zusammen gefasst.

---

```

FUNCTION CPL_Gatter
2  READ Eingänge
   READ Ausgänge
4  IF B = 1 AND Y = Z THEN Y := Z
   ELSE
6    IF B = 1 AND Y <> Z THEN Y := 0
       ELSE Y := A AND B
8    MAKE neue Saboteur-Eingangssignale
   END FUNCTION

```

---

Abbildung 4.8: Pseudo-Code der Unterbrechungsfehlermodellierung im CPL-Gatter

Beim Gebrauch einer 9-wertigen Logik ( $\rightarrow$ 4.1.3) wird die Modellierung noch präziser. In diesem Fall liegt der Bus mit Pull-up-Widerständen im hochohmigen Zustand nicht auf 'Z' sondern auf 'H', wenn die Widerstände an der Betriebsspannung liegen, oder auf 'L', wenn die Widerstände zur Masse führen. In diesem Fall wird das am Ausgang geschriebene 'Z' durch die Auflösungstabelle ( $\rightarrow$ Anhang A.1.4) richtig in 'H' oder 'L' zusammengefasst. Für das klassische Haftfehlermodell ( $\rightarrow$ 4.3.2.1) sind solche speziellen Betrachtungen beim CPL-Gatter nicht erforderlich, da sich keine Unterschiede im Verhalten ergeben.

### 4.2.3 Transfer-Gatter

Ein Transfer-Gatter (engl. *Transmission-Gate*) ist ein kompakter Schalter in CMOS-Technologie. Es entsteht durch eine Kombination eines PMOS- und eines NMOS-Transistors. Für NMOS-Transistoren ist es typisch, dass sie Low-Pegel gut leiten ( $\rightarrow$ 4.2.2), aber High-Pegel schlecht. Für PMOS-Transistoren verhält es sich umgekehrt. In der Abbildung 4.9 ist dieser Zusammenhang in einem Diagramm skizziert. Durch die Kombination beider Transistoren bekommt man einen Gesamtwiderstand  $R_{TG}$ , der sich aus der Parallelschaltung der beiden Transistorwiderstände ergibt. Für den leitenden Fall sieht das Schalterverhalten somit günstiger aus als bei einem Einzeltransistor.

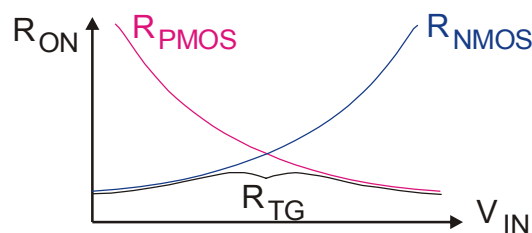


Abbildung 4.9: Widerstandsverhalten von NMOS- und PMOS-Transistor

Die Zusammenschaltung der beiden Transistoren ist in der Abbildung 4.10-a dargestellt.

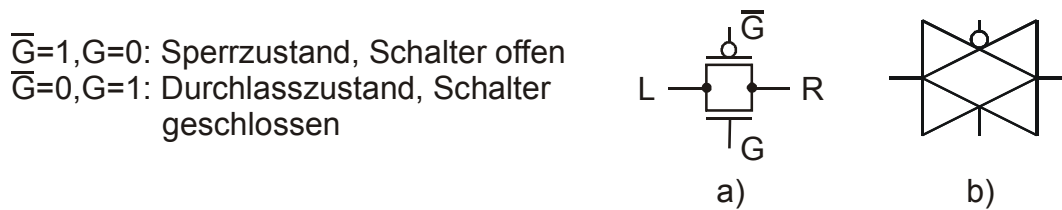


Abbildung 4.10: a) Zusammenschaltung und b) Symbol des Transfer-Gatters

Beim Transfer-Gatter wird der NMOS-Transistor mit dem Kontrollsignal  $G$  und der PMOS-Transistor mit dem invertierten Signal  $\bar{G}$  angesteuert. Ein High-Pegel an  $G$  bedeutet, dass der Schalter geschlossen ist. Ein Low-Pegel führt zum Sperren beider Transistoren. Damit öffnet der Schalter. Außerdem arbeitet ein Transfer-Gatter bidirektional, d. h., ein Signal kann sowohl von links ( $L$ ) nach rechts ( $R$ ) als auch umgekehrt übertragen werden.

In vielen HDL-Modellen, insbesondere bei VHDL-Beschreibungen, ist das Transfer-Gatter nur als unidirektionaler Schalter umgesetzt. In der Abbildung 4.11-b ist dies veranschaulicht. Zum einen liegt es darin begründet, dass einige Simulatoren mit der bidirektionalen Modellierung Probleme haben. Zum anderen genügt für viele funktionale Simulationen diese Vereinfachung. Die Abbildung 4.11-a soll die Ankopplung eines beliebigen Ausgangs an einen Bus über das Transfer-Gatter demonstrieren. In diesem Fall ist beispielsweise eine unidirektionale Übertragung für die funktionale Simulation völlig ausreichend. Das genauere bidirektionale Verhalten bleibt unberücksichtigt.

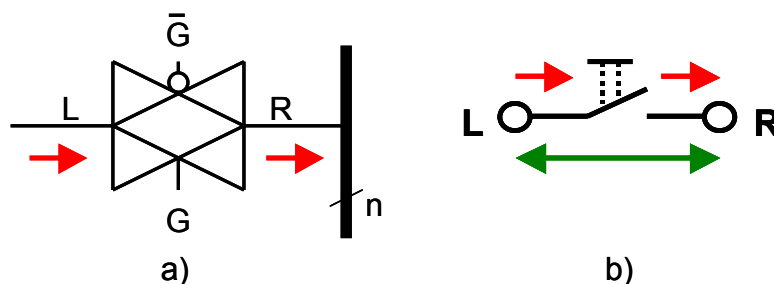


Abbildung 4.11: a) Bus-Anschaltung eines Transfer-Gatters und b) Schalterverhalten

Bei einer Simulation mit Fehlern bleiben besonders bei der einfachen Modellierung mögliche Probleme unentdeckt. Auf zwei mögliche Fehlerfälle, die speziell beim Transfer-Gatter auftreten, erfolgt im Folgenden eine nähere Untersuchung.

In dem einen Fall wird einer der Kontrolleingänge nicht invertiert angesteuert. Das Resultat ist beispielsweise, dass es zum Durchlassen nur eines Spannungspegel kommt. Für den anderen Pegel entsteht ein hochohmiger Zustand bzw. es wird ein Sperrzustand interpretiert.

Im zweiten Fehlerfall treffen zwei unterschiedliche Pegel von  $L$  und  $R$  im durchgesteuerten Transfer-Gatter aufeinander. Praktisch kann dies ohne Strombegrenzung zur Zerstörung des Gatters führen.

In der SystemC-Implementierung wurde auf diese Fehlerfälle derart Rücksicht genommen, dass vor der Signalauswertung und Ausgabe zunächst ein Lesevorgang von den Ein- und Ausgängen erfolgt und bei einer Störsituation eine spezielle Fehlervariable oder ein Fehlersignal gesetzt wird. Um den Aufwand hierfür gering zu halten, wurde auf eine zusätzliche Portimplementierung und Verdrahtung verzichtet. Zum einen bestand die Option, Signale und Variablen global zu erstellen, so dass alle Komponenten auf diese Zugriff haben. Zum anderen wurde die Fehlersignalisierung lokal implementiert. In diesem Fall war nur ein Zugriff aus der Testbench heraus möglich (→5.4.3), z. B. um das Signal in einer Trace-Datei aufzuzeichnen. Bei einigen Experimenten zeigte sich, dass bei vielen Zugriffen die Lösung mit dem Signal etwas schneller war als die Variante mit der Variablen.

Im Listing 4.5 ist die Implementierung eines Transfer-Gatters mit Fehlersignalisierung dargestellt.

---

```
2  SC_MODULE(tg1) {
    sc_in <sc_logic> G, _G, din;
4   sc_inout <sc_logic> dout;

6   sc_signal<sc_logic> sig_state;    //Signal für Fehler

8   void do_tg1();

10  SC_CTOR(tg1) {
    SC_METHOD(do_tg1);
12    sensitive << G << _G << din;
    . . .
14  . . .
    void tg1::do_tg1() {
16    if ((sel.read()==sc_logic_1) & (selnot.read() == sc_logic_0)) {
        dout.write(din); }
18    else {
        dout.write(sc_logic_Z); }
20    if (G.read() == _G.read()) {
        sig_state.write(sc_logic_1); }
22    else {
        sig_state.write(sc_logic_0);
24    . . .
```

---

Listing 4.5: Transfer-Gatter mit Fehlersignalisierung

In dem Beispiel wird ein Fehlersignal (Zeile 6) auf '1' gesetzt, falls die Kontrolleingänge nicht invertiert zueinander sind (Zeile 20). In den Zeilen 16-19 ist die normale Funktion des Transfer-Gatters realisiert.

Ein verfeinertes Modell des Transfer-Gatters erreicht man mit einer Implementierung auf der im Abschnitt 4.1.3 vorgeschlagenen Schalterebene in SystemC. Hierbei wird das Gatter durch die Transistorzusammenschaltung aus der Abbildung 4.10-a formuliert. Das Listing 4.6 zeigt einen Ausschnitt des Modell-Codes.

---

```
2  . . .
   #include "nmos.h"
   #include "pmos.h"
4
   SC_MODULE (tg) {
6       sc_inout<sc_logic> i0,r0;
       sc_in<sc_logic> ctrl, _ctrl;
8       nmos *nmos1;
       pmos *pmos1;
10
       SC_CTOR (tg) {
12           nmos1 = new nmos ("nmos1");
           nmos1->a0(i0);
14           nmos1->z0(r0);
           nmos1->ctrl(ctrl);
16
           pmos1 = new pmos ("pmos1");
18           pmos1->a0(r0);
           pmos1->z0(i0);
20           pmos1->ctrl(_ctrl);
       }
   . . .
```

---

Listing 4.6: Transfer-Gatter mit Transistoren der Schalterebene

Ein wesentlicher Vorteil des Schalterebenenmodells besteht nun in der zur Verfügung stehenden bidirektionalen Signalübertragung. Durch dieses Transfer-Gatter lassen sich z. B. auch Komponenten der konservativen reversiblen Logik modellieren [109], was mit vielen anderen Simulatoren nicht zu realisieren ist. Reversible Logik ist ein vielversprechendes Paradigma für vielleicht künftig bedeutungsvolle Computer [110]. So wird diese Logik z. B. für Quantenrechner [109], für optische Rechner [107] und für Low-Power-CMOS-Designs [108] vorgestellt. Erste Publikationen für Addierer- und PLA-Schaltungen mit reversibler Logik mit Hilfe von Feynman- und Fredkin-Gattern wurden bereits publiziert [110,111]. Im Anhang A.1.5 befindet sich das ausführliche Simulationsmodell eines CMOS-Fredkin-Gatters in SystemC.

### 4.2.4 Simulationsexperimente auf der Schalterebene

In einigen Experimenten wurde der Einfluss der Verfeinerung, der durch die Modellierung auf der Schalterebene entsteht, mit einigen Schaltungsmodellen in SystemC untersucht. In der Tabelle 4.3 sind die Ergebnisse wiedergegeben. Die dargestellten Zeiten sind in Bezug auf das entsprechende Gattermodell normiert. Die Schaltungen *TGB1* und *TGB2* sind Schiebeketten, die aus Transfer-Gattern bestehen.



Die Schaltung *MUX16* ist ein Multiplexer, der hauptsächlich aus Transfer-Gattern aufgebaut ist. Die Modelle *c17*, *MSAB* und *c432* sind kombinatorische Schaltungen.

Entwurfsebene	Simulationszeiten (normiert)					
	TGB1	TGB2	MUX16	c17	MSAB	c432
Gatterebene	1	1	1	1	1	1
Schalterebene	1,07	1,54	1,05	1,82	4,16	7,51

Tabelle 4.3: Simulationszeiten von Modellen der Gatter- und Schalterebene

Es zeigt sich, dass für die untersuchten Modelle der zeitliche Mehraufwand an Simulationszeit sich in einem Rahmen bewegt, der eine Implementierung auf der Schalterebene für eine Entwurfsverfeinerung akzeptabel und durchführbar erscheinen lässt. Wobei für kleine Schaltungen der Overhead an Rechenzeit geringer ausfällt als für größere.

## 4.3 Fehler und deren Modellierung in SystemC

Bei einem elektronischen System kann während seines Lebenszyklus eine Reihe von Fehlern auftreten. Dies ist schon in der Spezifikationsphase möglich und auch im späteren Einsatz nicht ausgeschlossen. Die Abbildung 4.12 zeigt den Zusammenhang zwischen den Fehlern und ihrem Auftreten bezüglich des Lebenszyklus eines Systems [151].

Die Auswirkungen des Auftretens von Fehlern auf den Systemzustand sind in den Abschnitten 5.1.2 und 5.1.3 beschrieben. In diesem Abschnitt werden die Ursachen von Fehlern aufgeführt und diverse Implementierungen in Fehlermodelle als SystemC-Beschreibungen vorgestellt. Fehleruntersuchungen der Spezifikations- und Designphase sind dabei nicht im Fokus dieser Arbeit. Außerdem treten in den meisten der heutigen Systeme mögliche Fehler in Hardware und in Software auf. Bezüglich des Auftrittsortes erfolgt eine Beschränkung auf Hardware-Modelle.

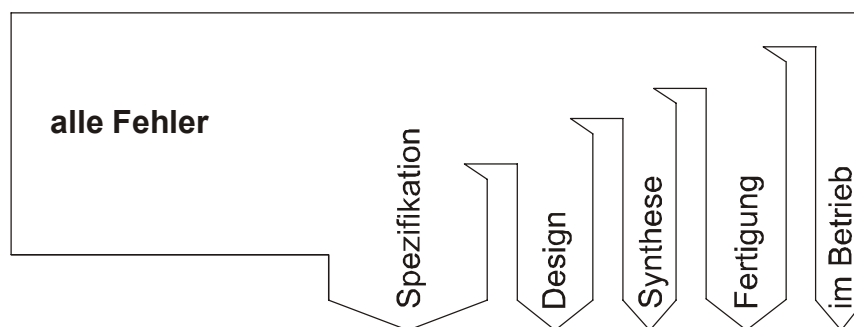


Abbildung 4.12: Fehler im Lebenszyklus eines Systems

### 4.3.1 Fehlertypen und Fehlerursachen

Als Fehler (engl. *faults*) bezeichnet man das Verhalten eines Systems oder einer Schaltung, bei dem bestimmte Systemgrößen von den zu erwartenden Werten abweichen [8]. Diese Abweichungen betreffen dabei das elektrische und/oder logische Verhalten. Ein Fehler kann man dabei bezüglich seiner Natur, seines Wertes, seines Ausmaßes und seiner Dauer charakterisieren [154]. Das Fehlerverhalten lässt sich in unterschiedliche Fehlertypen unterscheiden. In der Abbildung 4.13 ist eine Klassifizierung bezüglich der Dauer dargestellt.

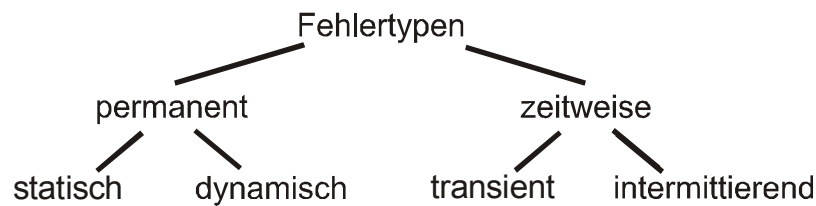


Abbildung 4.13: Eine Klassifizierung von Fehlern

Fehler können permanent von statischer oder dynamischer Natur sein oder nur vorübergehend als transiente bzw. intermittierende Versionen auftreten. Permanente Fehler werden zuweilen auch als harte Fehler (*hard faults*) und die zeitweisen (temporären) Fehler als weiche Fehler (*soft faults*) bezeichnet [92,175].

- Ein **statischer Fehler** ist unabhängig von der Schaltungsfrequenz und reproduzierbar. Er lässt sich meist besser testen als andere Fehlertypen. Ein solcher Fehler kann z. B. das Haften eines Schaltungsknotens auf einen festen Spannungspegel sein.
- Ein **dynamischer Fehler** tritt abhängig vom dynamischen Verhalten der Schaltung auf. Er erscheint häufig erst ab einer bestimmten Schaltgeschwindigkeit oder innerhalb eines Frequenzbandes. Ein typischer Vertreter hierfür ist der Verzögerungsfehler.
- Ein **transienter Fehler** tritt im System auf, ohne dass dazu ein Hardware-Defekt vorliegt. Ursachen für einen solchen Fehler sind Umwelteinflüsse wie  $\alpha$ -Partikel-, Neutronen- und kosmische Strahlung [160], elektromagnetische Entladung oder elektromagnetische Kopplung. Solche Fehler können zum Beispiel zu sporadisch auftretenden Abweichungen der Spannungspegel (*overshoots, glitches*) führen.
- Ein **intermittierender Fehler** tritt auch vorübergehend und mit eher zufälligem Erscheinungsbild auf. Die Ursachen entstammen typischerweise aus dem System selbst und entstehen z. B. durch Hardware-Komponenten, die im Grenzbereich arbeiten. Typische Vertreter von solchen Fehlern sind Spannungsimpulse oder Spannungseinbrüche (*glitches*) auf Signalleitungen.

Mitunter treten sie verstärkt in Gruppen, in sogenannten *bursts* auf. Ebenso führen Wackelkontakte und kalte Lötstellen zu diesem Fehlertyp.

Ca. 80 % bis 90 % aller auftretenden Fehler in Hardware sind transienter oder intermittierender Natur [156]. Neben den hier vorgestellten Klassifizierungen von Fehlern findet man noch einige andere in der Literatur. Eine sehr umfangreiche Aufstellung mit 256 unterschiedlichen Kombinationsklassen ist in [155] wiedergegeben.

Ein Begriff, welcher bei Fehleruntersuchungen in Hardware auftritt, ist der Begriff des *Defektes*. Ein Defekt (engl. *defect*) kann die physikalische Ursache eines Fehlers sein. Allerdings führt nicht jeder Defekt innerhalb einer Schaltung unbedingt zu einem Fehler. Häufig äußern sich Defekte zunächst nur in einer leichten Abweichung von einigen Schaltungsparametern, die sich durch Alterungsprozesse verstärken und später zu einem Fehler führen. So ist es durch unsachgemäßen Umgang möglich, einen Schaltungsteil vorzuschädigen, wie beispielsweise durch das Berühren von Eingängen einer Schaltung mit MOS-Transistoren (z. B. ein Speichermodul) ohne ESD-Schutzmaßnahmen. Ein Schaltungsausfall tritt dann aber oftmals erst nach Monaten oder Jahren auf.

Defekte als Fehlerursachen sind eine physikalische Abnormität und haben ein konkretes Erscheinungsbild. Jedoch lassen sie sich selten direkt erkennen. Zum einen liegt das an den geringen geometrischen Dimensionen und zum anderen daran, dass sie durch nachfolgende Schichten verdeckt werden. Der Nachweis eines Defekts ist oft erst durch dessen Auswirkung auf die elektrischen Eigenschaften der Schaltung möglich [166]. Die Entstehung von Defekten kann viele Ursachen haben. Beispielhaft sind einige davon aufgeführt [8]:

- Fehlstellen im Ausgangsmaterial (statistisch verteilt);
- Herstellungsfehler, z. B. durch ungleichmäßiges Ätzen oder Dotieren, zusätzliche Partikel durch Verschmutzung, Justierungsfehler oder Schäden an den Masken, Fehler beim Bonden;
- Probleme nach der Herstellung, z. B. statische Aufladung bei CMOS, thermische oder elektrische Überlastung, Atommigration, Korrosion.

Mittlerweile macht in modernen Halbleiterschaltungen, wie z. B. in FPGAs, das Verbindungsnetzwerk einen Großteil der Fläche aus und stellt auch eine wesentliche Fehlerquelle dar. In der Abbildung 4.14 sind einige Defekte bzw. Fehler geometrisch dargestellt. Zusätzliches Leitungsmaterial (a) und fehlende Isolation (c) führen in ungünstigen Fällen zu Kurzschlüssen (engl. *shorts*). Fehlendes Material (b) oder zusätzliche Isolation führen zu Unterbrechungen (engl. *opens*). Ein Großteil (ca. 90 %) der zu erwartenden, nachweisbaren Fehler in integrierten Schaltungen ist von einem dieser Typen oder entsteht durch eine Kombination dieser [157].

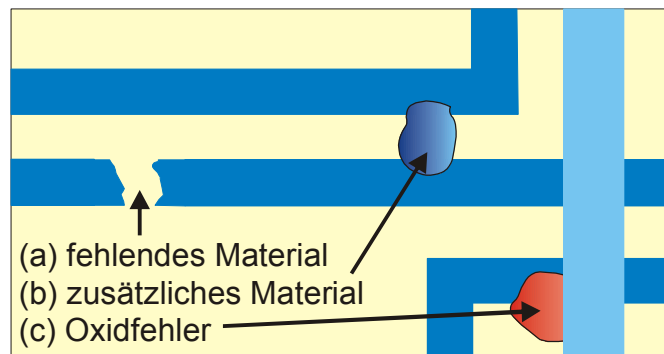


Abbildung 4.14: Geometrische Fehler durch zu viel und zu wenig Material

Außer in den Verbindungsstrukturen treten Defekte in den Transistoren auf. Transistor-Fehler werden häufig durch Parameterfehler verursacht [151]. In MOS-Transistoren wird das Siliziumoxid oftmals als Isolationsmaterial und als Gate-Oxid verwendet. Aus diesem Grund sind Oxidfehler häufig die Ursache von Transistor-Fehlern. Ist z. B. das Gate-Oxid zu dünn, so ist das Bauelement anfällig für eine Schädigung am Gate-Anschluß durch eine Überspannung. Zuviel Gate-Oxid kann wiederum die Schwellspannung verschieben und so das Schaltverhalten beeinflussen.

### 4.3.2 Fehlermodelle

Ein Fehlermodell bildet die Auswirkungen eines Defektes auf elektrische oder logische Eigenschaften ab. Durch zugrunde liegende Annahmen folgt, welche Defekte durch ein Fehlermodell abgedeckt werden. Mit Hilfe geeigneter Testmuster lassen sich modellierte Fehler nachweisen und eine Schlussfolgerung hinsichtlich des Auftretens eines Defektes zu. Der Rückschluss eines Fehlers auf einen bestimmten ursächlichen Defekt ist deutlich schwieriger, da verschiedene Defekte die gleichen elektrischen Auswirkungen haben. Ein Fehlermodell deckt somit häufig verschiedene Defekte ab. So sind z. B. mit dem Haftfehlermodell (→4.3.2) nicht nur Kurzschlüsse zu den Versorgungsspannungen, sondern auch Überbrückungen zwischen Signalleitungen auffindbar.

Fehlermodelle für integrierte Schaltungen sind immer mit einer Abstraktionsebene (→2.1) verknüpft, für die sie definiert sind. Um die Auswirkungen von Fehlern, die auf einer niedrigen Abstraktionsebene angenommen werden, auf höheren Ebenen zu untersuchen, gibt es die Methode der induktiven Fehleranalyse [12]. Die induktive Fehleranalyse versucht für eine Schaltungstechnologie und einem Schaltungslayout, eine realistische und gewichtete Menge an Fehlern zu spezifizieren. Weiterhin fließen Statistiken zu auftretenden Defekten mit ein. Ergebnis ist eine Liste von Fehlern, die nach der Auftrittswahrscheinlichkeit sortiert ist. Die Zielmenge der Fehler (Kurzschlüsse, Unterbrechungen, etc.) führt zu diversen Fehlermodellen in den Abstraktionsebenen des Entwurfs.

Die Tabelle 4.4 zeigt in Anlehnung an [12] eine mögliche Zusammenstellung von Fehlermodellen zu den jeweiligen Abstraktionsebenen.

Abstraktionsebene	Implementierungs-abhängig	Technologie-abhängig	Fehlermodell
Architektur	nein	nein	allgemeine Funktionsfehler
Register-Transfer	nein	nein	Graphen, funktionale Blöcke
Gatter	ja	nein	Haft-, Verzögerungs-, Brückenfehler
Schalter	ja	ja	stuck-on, stuck-open
Elektrische Schaltung	ja	ja	Parameterfehler, „Hard-“, „Soft-“Defekte
Layout	ja	ja	Spot-Defekte

Tabelle 4.4: Abstraktionsebene und Fehlermodelle

Die Aufstellung in der Tabelle 4.4 erhebt keinen Anspruch auf Vollständigkeit. Die Entwicklung neuer, effektiver Fehlermodelle bleibt Gegenstand aktueller Forschungsaktivitäten. Insbesondere durch die Weiterentwicklung bestehender und dem Aufkommen neuer Technologien gibt es einen Bedarf [34]. Jedoch existieren schon seit Jahren eine Reihe von Fehlermodellen, die beim Test integrierter Schaltungen besonders häufig Verwendung finden. Im Folgenden werden diese Fehlermodelle vorgestellt und Vorschläge zu deren Implementierung in SystemC unterbreitet.

#### 4.3.2.1 Haftfehlermodell

Das Haftfehlermodell (engl. *stuck-at fault model*) gehört zu dem am weitesten verbreiteten Fehlermodell. Dieses Modell bezeichnet man deshalb auch als das klassische Fehlermodell. Es wurde bereits in den 50er Jahren entwickelt [161] und ist das erste Modell, welches tiefgehend untersucht wurde [158]. Eine Vielzahl von Defekten lässt sich mit diesem Modell beschreiben, in CMOS-Technologie sind es bis zu 64 % [159]. Ein weiterer Grund für die weite Verbreitung dieses Modells liegt in der guten Möglichkeit, es in Simulationen und in der ATPG einzusetzen, da es sich einfach modellieren lässt. Zudem ist es weitgehend Technologie-unabhängig [162].

Das Haftfehlermodell basiert auf der Annahme, dass ein Eingang, Ausgang oder eine Leitung auf einem festen Wert liegt, so dass kein Signalwechsel möglich ist. Es gehört zur Gruppe der statischen Fehlertypen und wird überwiegend bei der Nutzung der Gatterebene verwendet. Man trifft es jedoch auch auf der RT-Ebene und Schalterebene [151] an und es finden sich einige Fehlermodelle auf anderen Abstraktionsebenen, mit denen es gut übereinstimmt [150] (→5.3.1.1). Dies liegt darin begründet, dass die

beiden Fehler stuck-at-1 und stuck-at-0 mit booleschen Variablen abzubilden sind. Das Haftfehlermodell lautet:

$$sa_x, x \in \{0,1\}$$

Ein stuck-at-0-Fehler (sa0) bedeutet, dass der entsprechende Knoten ständig auf logisch '0' liegt und ein stuck-at-1-Fehler (sa1) drückt aus, dass der zugehörige Knoten ständig auf logisch '1' gesetzt ist.

In der Abbildung 4.15 ist eine Haftfehlermodellierung an einem NAND-Gatter beispielhaft wiedergegeben. Die Kurzschlüsse  $F1$ ,  $F2$  und  $F3$  die im Transistormodell (b) zum Low-Pegel  $Gnd$  führen, werden in der Gatter-Beschreibung zu stuck-at-0-Fehlern. Die Gesamtzahl der zu modellierenden Fehler ergibt sich aus der Anzahl der verschiedenen Leitungen  $n$ . Werden nur einzeln auftretende Haftfehler angenommen, so beträgt deren Fehleranzahl  $2 \cdot n$ . Kommt es zusätzlich zur Betrachtung von Mehrfachfehlern, d. h., es sind zwei oder mehr Leitungen fehlerhaft, so steigt die Anzahl der Möglichkeiten auf  $3^n - 1$ . Eine Mehrfachfehlerbetrachtung erhöht allerdings kaum die Erkennung von Defekten gegenüber dem Einzelfehlermodell.

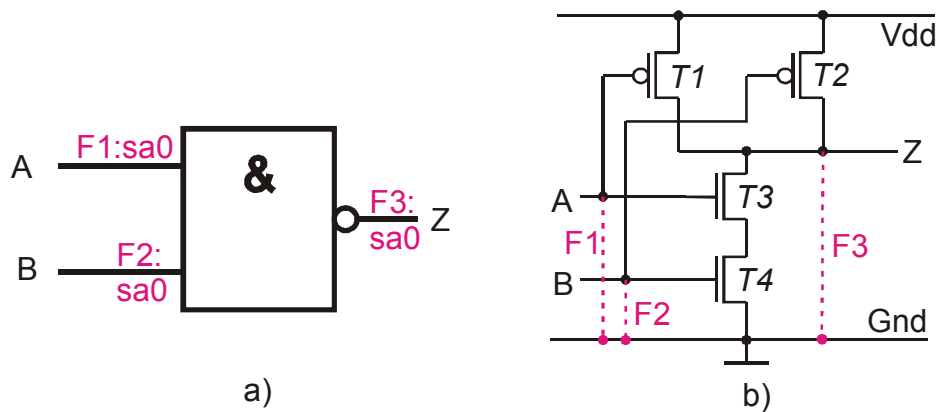


Abbildung 4.15: Haftfehlermodellierung a) Gatterebene und b) Transistorebene

Möchte man eine vollständige Simulation mit Haftfehlern durchführen, so müssen nicht an allen Pins oder Schaltungsknoten Fehler eingebaut werden. Aufgrund von Zusammenhängen kann man einige Fehler aus der Liste der zu simulierenden streichen. Gründe hierfür sind:

- Äquivalenz bzw. Nachweisidentität,
- Dominanzverhalten,
- Fehlerredundanz.

In der Tabelle 4.5 sind die Ergebnisse für das Beispiel aus der Abbildung 4.15 dargestellt. Die Zusammenhänge für das Fehlerverhalten werden im Folgenden anhand dieser kurz erläutert.

Zwischen zwei oder mehr unterschiedlichen Fehlern besteht *Äquivalenz*, wenn das Verhalten der Schaltung bei diesen identisch ist. Somit lassen sich diese Fehler zusammenfassen. Im Beispiel ergeben die beiden Fehler  $F1$  und  $F2$  das gleiche Ergebnis. Erkennbar ist dies an den beiden Ausgaben an  $Z(F1)$  und  $Z(F2)$  für den Ausgang  $Z$ .

Ein Fehler  $a$  besitzt *Dominanz* gegenüber einem Fehler  $b$ , wenn jeder Test, der Fehler  $b$  erkennt, auch Fehler  $a$  erkennt [7]. Im Beispiel in der Abbildung 4.15 besitzt der Fehler sa0-Fehler  $F3$  Dominanz gegenüber den sa1-Fehlern  $F1$  und  $F2$ .

Ein Fehler wird als *redundant* bezeichnet, wenn sich dieser durch keine Eingangsbelegung nachweisen lässt. Solche Fehler entstehen innerhalb der Schaltung z. B. dadurch, dass nicht benötigte Eingänge auf feste Pegel gelegt werden oder durch *rekongvergente Auffächerungen* [157]. Normalerweise ist es sinnvoll, solche Fehler von vornherein aus der Simulation auszuschließen. Allerdings gestaltet sich die Feststellung solcher Fehler oftmals schwieriger, als alle Fehlerfälle zu simulieren.

A	B	Z	Z(F1)	Z(F2)	Z(F3)
0	0	1	1	1	0
1	0	1	1	1	0
0	1	1	1	1	0
1	1	0	1	1	0

Tabelle 4.5: Funktionstabelle für NAND und Fehlermodelle

Anhand von Äquivalenz und Dominanz kann die Anzahl der zu betrachtenden Fehler deutlich reduziert werden. In eigenen Anwendungen konnte durch ein Werkzeug, basierend auf die in [32] beschriebene Anwendung, die Liste der zu simulierenden Fehler um ca. 40 % - 50 % gesenkt werden.

Die Umsetzung des Haftfehlermodells in SystemC ist relativ einfach, da es sich anhand boolescher Werte abbilden lässt. Z. B. kann in Abhängigkeit eines zusätzlichen Fehleraktivierungssignals bzw. -eingangs bei einer Komponente der Fehler ausgelöst werden. Die Abbildung 4.16 zeigt hierzu das SystemC-Modul schematisch.

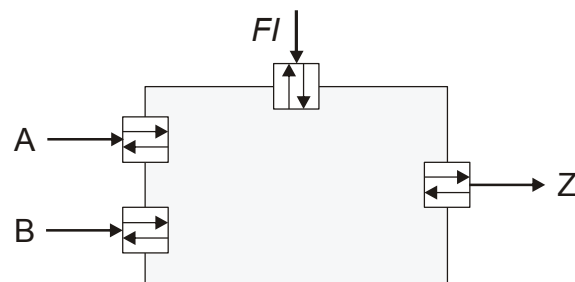


Abbildung 4.16: SystemC-Modul für Haftfehlermodellierung

Eine Lösung zeigt das Listing 4.7. In diesem Beispiel wird in Abhängigkeit des Eingangs *FI* ein Haftfehler in den Ausgang *Z* injiziert (Zeile 3, 7). Die Auswertung der Fehleraktivierung erfolgt mit bedingten Anweisungen in Form von *if-then*-Konstrukten. Andere bedingte Anweisungen, wie z. B. *switch-case*-Anweisungen, sind ebenso möglich. Für die meisten Fälle ist eine Fehlerinjektion an den Ausgängen ausreichend, da so bis auf die Eingänge alle Signalleitungen berücksichtigt werden. Ist eine erweiterte Fehlerinjektion in die Eingänge erforderlich, so kann man das Beispiel im Listing 4.7 dahingehend erweitern.

---

```
2  if (FI.read() == 1) {  
    Z.write(sc_logic_1);  
4  }  
    else {  
6      if(FI.read() == 2) {  
        Z.write(sc_logic_0);  
8      }  
    else {  
10         Z.write(...);  
12     }  
    }
```

---

Listing 4.7: Haftfehlermodellierung in SystemC

Eine andere Möglichkeit der Injektion von Haftfehlern besteht, neben dem Gebrauch von bedingten Anweisungen, durch den Gebrauch logischer Operationen. In Simulatoren, die nicht in der Lage sind, eine HDL-Beschreibung zu interpretieren, ist die Verwendung von Konstrukten einer Programmiersprache ausgeschlossen. In solchen Simulatoren hat sich die Verwendung von AND- und OR-Funktionen zur Fehlerinjektion angeboten [22]. In der Abbildung 4.17 ist dies für die Injektion eines stuck-at-1-Fehlers dargestellt. Erfolgt die Aktivierung der Fehlerinjektion  $FI = '1'$  über das OR-Gatter, so entsteht auf der Signalleitung *A'* ein stuck-at-1-Fehler. Mit dem Einsatz eines AND-Gatters anstelle des OR-Gatters und der Injektion  $FI = '0'$  entsteht ein stuck-at-0-Fehler.



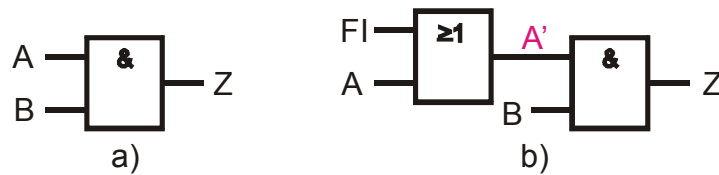


Abbildung 4.17: Stuck-at-1-Fehler mit OR-Gatter: a) Originalmodell, b) modifiziert

In SystemC ist die Injektion mit Hilfe logischer Operationen auch gut zu implementieren. Das Listing 4.8 zeigt exemplarisch die Injektion des stuck-at-1-Fehlers in Anlehnung an das Beispiel aus der Abbildung 4.17.

---

```
1  . . .
2  var_in = A.read();
   var_sal = FI.read();
4  var_out = (var_out | var_sal);    //sal
   Z.write(var_out);
6  . . .
```

---

Listing 4.8: Haftfehlermodellierung in SystemC mit arithmetischer Operation

#### 4.3.2.2 Brückenfehlermodell

Der Brückenfehler (engl. *bridging fault*) wurde 1974 als nicht erwünschte Verbindung zwischen zwei Leitungen vorgestellt [163] und zählt wie das Haftfehlermodell zu den statischen Fehlern. Die Überbrückung zwischen den Leitungen wird durch einen Widerstand mit einem niedrigen Wert hervorgerufen. Bei einem Widerstandswert von null entsteht ein idealer bzw. harter Kurzschluss. Für MOS-Technologien ist allerdings ein Brückenwiderstand im Bereich einiger  $k\Omega$  verhältnismäßig niederohmig. Der Überbrückungsfehler lässt sich in drei Kategorien einteilen:

- Überbrückung innerhalb eines logischen Elements,
- Überbrückung zwischen zwei Leitungen bzw. Schaltungsknoten ohne Rückkopplung,
- Überbrückung zwischen zwei Leitungen bzw. Schaltungsknoten mit Rückkopplung.

Der *Brückenfehler innerhalb eines Logikelementes* lässt sich auf der Gatterebene schwer beschreiben, da hierfür die Fehler mindestens auf der Schalterebene, z. B. die Überbrückungen zwischen den Transistoren, näher untersucht werden müssen. Solche Brückenfehler bzw. Defekte werden meist auch von anderen Fehlermodellen wie dem Haftfehlermodell (→4.3.2.1) oder dem stuck-on-Modell abgedeckt (→4.3.2.3).

Der *Brückenfehler zwischen zwei Leitungen* ist derjenige, welcher üblicherweise simuliert wird.

Der *Brückenfehler zwischen zwei Leitungen*, welcher eine *Rückkopplung* in der Schaltung bewirkt, ist gesondert zu betrachten. Bei diesen Fehlern wird aus einem

kombinatorischen Schaltungsteil einer mit sequenziellen Verhalten. In Schaltungen mit einer ungeraden Anzahl von Invertierungen bewirken diese Rückkopplungen zudem Oszillationen. Für einen Simulator stellt eine solche Brücke ein Problem dar. Bei der Signalberechnung entstehen bei einem Rechenschritt immer wieder neue Ereignisse in der abzuarbeitenden Ereigniswarteschlange, so dass die Simulation üblicherweise in einem Endlos-Zyklus hängen bleibt, falls eine Schaltungsüberprüfung auf dieses Problem nicht schon vorher hingewiesen und die Simulation verweigert hat. Abhilfe ist möglich, wenn im Rückkopplungszweig ein Modul (*loop breaker*) zwischengeschaltet wird, welches die Rückkopplung besonders berücksichtigt [221]. Führt man beispielsweise eine zusätzliche Verzögerung in dem Rückkopplungsmodul ein, so ist eine Taktfortschaltung möglich und der Endloszyklus wird umgangen.

Ein weiteres Problem besteht im Umfang der anzunehmenden Fehler bzw. in der sinnvollen Generierung der Anzahl an zu simulierenden Brückenfehlern. Zum einen wird die vollständige Modellierung schon bei Schaltungen mit einigen hundert Leitungen sehr umfangreich, wenn jede Leitung mit jeder einen Kurzschluss führen darf. Zum anderen ist eine solche erschöpfende Fehlergenerierung wenig sinnvoll, da für einen Brückenfehler als Resultat eines Defekts Nachbarschaftsbeziehungen nötig sind. Einfache Nachbarschaftsverhältnisse ergeben sich bei einem Gatter z. B. mit den Eingängen. Für Schaltungsknoten von unterschiedlichen Logikelementen sieht es schon schwieriger aus, da hierzu Platzierungsinformationen aus der Geometrieebene nötig sind (→2.1).

Frühe Brückenfehlermodellierungen waren so, dass sie eine Wired-OR- und Wired-AND-Verknüpfung der beteiligten Leitungen bildeten [163]. Die Abbildung 4.18 soll den Wired-OR-Fehler verdeutlichen.

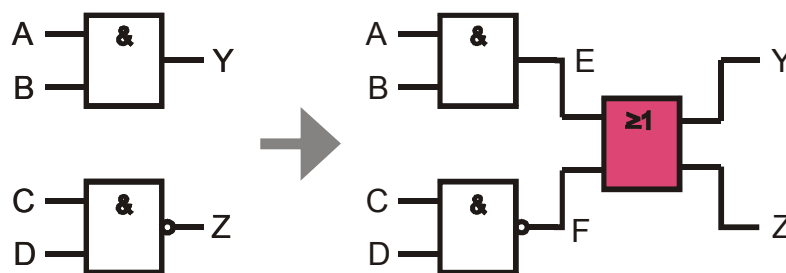


Abbildung 4.18: Wired-OR-Brückenfehler

Ein Brückenfehler bleibt unerkannt, solange die beteiligten Leitungen den gleichen Logikwert führen. Um den Fehler also zu provozieren und aufzudecken, müssen die Leitungen unterschiedliche Werte haben. Beim Wired-OR-Modell dominiert die logische '1' und beim Wired-AND-Modell die logische '0'. Die Verknüpfung ergibt, dass die Werte beider Leitungen gleich sind. Die entsprechende Umsetzung in SystemC ist im Listing 4.9 wiedergegeben.

---

```
. . .
2  int v_fi=fi.read();
    int temp;
4   switch (v_fi) {
        case 0: Y.write(E.read());    //normale Durchschaltung
6           Z.write(F.read());
              break;
        case 1: temp = E.read() & F.read();    //And-Brücke
              Y.write(temp);
10             Z.write(temp);
              break;
        case 2: temp = E.read() | F.read();    //Or-Brücke
              Y.write(temp);
14             Z.write(temp);
              break;
16     default: Y.write(E.read());    //normale Durchschaltung
        . . .
```

---

Listing 4.9: Wired-Fehler-Modul in SystemC

In Abhängigkeit eines Kontrollwertes  $f_i$  wird in einem Brückenfehlermodell eine OR-Operation für die beiden Ausgänge  $E$ ,  $F$  (Zeilen 12) oder eine AND-Operation getätigt (Zeile 8). In den anderen Fällen ist kein Fehler vorhanden.

In CMOS-Schaltungen ist das Wired-OR- bzw. Wired-AND-Modell häufig ungeeignet, einen Brückenfehler zu modellieren. Hierbei ergibt sich ein Spannungswert, der nicht in allen Fällen einer logischen '0' oder '1' zugeordnet werden kann, bedingt aus den Widerstandsverhältnissen des NMOS- und PMOS-Transistornetzwerks. Um hierfür eine genauere Modellierung zu erreichen, wurde in [196] ein *voting model* vorgestellt. Jedoch zeigte auch diese Modellierung ihre Schwächen. So war sie z. B. nicht in der Lage das *Byzantine General's Problem* zu lösen [170]. Dieses Problem entsteht dadurch, dass verschiedene Gatter den gleichen Wert einer Analogspannung in unterschiedliche digitale Werte interpretieren können. Dieser Umstand basiert auf den Schwellwertspannungen an den Gate-Eingängen, die durchaus schwanken können. Die hinreichend genaue Modellierung des Brückenfehlermodells war Gegenstand weiterer Arbeiten [167,168,172] und bleibt es sicher auch für künftige. So gibt es z. B. unterschiedliche Bewertungen der Leitungen bezüglich der Signalpriorität, dem sogenannten *Aggressor-Victim*-Modell [101]. Zur Modellierung dieses Modells genügt eine kleine Änderung im Listing 4.9, beispielsweise brauchen hierzu nur die beiden Ausgangswerte den Wert des Aggressors bekommen.

In vielen Fällen wird versucht, eine genaue analoge SPICE-Modellierung und -Simulation [141] durch eine schnellere diskrete Berechnung zu ersetzen, z. B. mit der Verwendung mehrerer Signalstärken. Da SystemC keinen Datentypen mit einem großen Vorrat an verschiedenen Signalstärken bietet, fand in Anlehnung an [12] für das Brückenfehlermodell das logische 'X' Verwendung. Das Listing 4.10 zeigt hierzu kurz die Implementierung.

---

```
. . .
2  case 2: Y.write(sc_logic_X);    //Indetermination 'X'
      Z.write(sc_logic_X);
4      break;
. . .
```

---

Listing 4.10: Unbestimmt-Fehler-Modul in SystemC

Für den Test von Brückenfehlern in CMOS-Schaltungen hat sich in den letzten Jahren ein weiteres Vorgehen etabliert. Schaltungen in CMOS besitzen die Eigenschaft, dass sie nahezu keinen Ruhestrom haben. Andererseits bedingen Brückenfehler in CMOS-Schaltungen einen Querstrom, der sich in seinem Wert vom Ruhestrom deutlich unterscheidet. Diese Eigenschaft wird mit Hilfe passender Testmuster und dem Überwachen des Betriebsstroms genutzt, um solche Brückenfehler in sogenannten IDDQ-Tests aufzuspüren [164,166]. Zur Validierung der Muster in einem SystemC-Modell ist eine Erweiterung des Brückenfehlermodells um die zusätzlichen Eigenschaften des Stromverbrauchs zu erweitern. Die dazu nötigen Werte des Stromverbrauchs können z. B. aus einer induktiven Fehleranalyse (→4.3.2) oder einer SPICE-Simulation [141] stammen. In [171] wird zur Kostenmodellierung und -optimierung in SystemC eine Klasse *cost* vorgestellt. In dieser Klasse sind unter anderem neben einer Identifikation, dem Kostenwert auch Grenzwerte etc. realisiert. Durch die Verwendung einer Klasse mit ähnlichen Methoden und Elementen ließen sich Brückenfehlermodelle bezüglich des Stromverbrauchs modellieren und auswerten.

Für Halbleiterschaltungen, die in neuesten *deep-submicron*-Prozessen gefertigt sind, erweist sich der IDDQ-Test zunehmend schwierig. Bedingt durch die kleinen Strukturen, z.T. sind Schichtdicken von wenigen Atomlagen in Gebrauch, wächst der Anteil an Leckströmen (*leakage*) in der Schaltung. Der Unterschied eines fehlerhaften Stroms durch einen Brücken-Defekt zum Strom ohne Fehler schwindet, so dass sich dieser nicht mehr brauchbar messen lässt.

### 4.3.2.3 Transistorfehlermodelle

Auf der Schalterebene, die in Halbleiterschaltungen vereinfacht durch Transistoren dargestellt wird, gibt es einige Fehlermodelle, die eine genauere Fehleruntersuchung erlauben. Zwei typische Fehlermodelle sind das *stuck-on*- und das *stuck-off*-Fehlermodell.

Das *stuck-on*-Fehlermodell bedeutet einen stets leitenden Transistor, unabhängig vom Wert des Gate-Anschlusses. Für den *stuck-off*-Fehler gilt der umgekehrte Fall, dieser ist unabhängig vom Wert am Gate-Pin stets gesperrt.

Bisher war eine Modellierung dieser Fehlermodelle in SystemC nicht lösbar, da für diese Entwurfsebene die Voraussetzungen fehlten. Im Abschnitt 4.1.3 wurde jedoch die Modellierung von SystemC-Beschreibungen auf der Schalterebene vorgestellt. Mit Hilfe der dort vorgestellten Erweiterungen lässt sich nun auch das Verhalten der beiden Transistorfehler nachbilden. Das Listing 4.11 zeigt die Erweiterung des einfachen

NMOS-Transistor-Modells ( $\rightarrow$ Listing4.3) für den *stuck-on*-Fehler. In Abhängigkeit des Fehlerkontrolleingangs  $fi$  wird der stuck-on-Fehler aktiviert (Zeile 29). Erfolgt ein Aufruf des Modells bei einem  $fi$ -Wert von ungleich null, so kommt es zur Ausgabe des Fehlerwertes nach der Tabelle *nmosTabFi*. Andernfalls erfolgt die korrekte Ausgabe gemäß dem Listing 4.3.

Ähnlich sieht die Umsetzung für den *stuck-off*-Fehler aus. Es werden die Werte so ausgegeben, als wenn immer eine logische '0' am Gate-Anschluss wäre.

Ein dritter Fehler, welchem bei der Modellierung auf der Schalterebene ein besonderer Stellenwert zukommt, ist der *stuck-open*-Fehler. In einigen Publikationen findet man ihn auch unter der Bezeichnung *open-line* oder *line-break* wieder.

In bipolaren Schaltungen lässt sich ein *stuck-open*-Fehler ausreichend gut mit einem Wired-AND- oder Wired-OR-Modell nachbilden, ähnlich dem Brückenfehlermodell im Listing 4.9.

---

```
2  SC_MODULE (nmos) {
    sc_out<sc_logic> z0;
4   sc_in<sc_logic> a0, ctrl;
    sc_in<int> fi;
6   sc_logic a0_l, ctrl_l, res_l;
    int a0_t, ctrl_t, res_t;
8
    void doit();
14 . . .

```

---

```
16 const char nmosTabFi[4][4] = {
    //      0      1      Z      X
18     { '0', '0', '0', '0' },      //0
    { '1', '1', '1', '1' },      //1
20     { 'Z', 'Z', 'Z', 'Z' },      //Z
    { 'X', 'X', 'X', 'X' },      //X
22 };

24 void nmos::doit() {
    a0_l = a0.read();
26     a0_t = (int) a0_l.value() ;
    ctrl_l = ctrl.read();
28     ctrl_t = (int)ctrl_l.value();
    if (fi.read())
30         z0.write((sc_logic)nmosTabFi[a0_t][ctrl_t]);
    else
32     . . .

```

---

Listing 4.11: Erweiterung des NMOS-Transistors mit stuck-on-Fehler

In CMOS-Schaltungen fließt im Ruhefall so gut wie kein Strom. Außerdem besitzen Schaltungsknoten in der Schaltung immer gewisse parasitäre Kapazitäten, die in der Lage sind, einen Teil der elektrischen Ladung zu speichern. Werden nun solche Kapazitäten von einem Teil der Schaltung abgekoppelt, wie z. B. durch einen *stuck-*

*open*-Fehler, so bekommt die Komponente ein zusätzliches Speicherelement. Ein kombinatorisches Schaltungsteil wird so zu einem sequenziellen. Aufgrund dieses Verhaltens ist eine besondere Betrachtung eines *stuck-open*-Fehlers innerhalb eines Logikelementes, welches aus MOS-Transistoren besteht, von Bedeutung. Da ein *stuck-open*-Fehler eine hochohmige Unterbrechung ausdrückt, bietet sich in SystemC der Gebrauch des Wertes 'Z' an. Im Listing 4.12 ist dies dargestellt. Die speichernden Eigenschaften selbst werden vom Transistormodell berücksichtigt (→Tab.4.1).

---

```
2      . . .
   int v_fi=fi.read();
   switch (v_fi) {
4       . . .
       case 3:  Z.write(sc_logic_Z); //stuck-open
6               break;
       default: Z.write(A.read());  //normale Durchschaltung
8               break;
   . . .
```

---

Listing 4.12: Einfacher stuck-open-Fehler in SystemC

Der Einsatz des *stuck-open*-Fehlermodells aus dem Listing 4.12 oberhalb der Schalterebene ist damit auch möglich, jedoch gilt hierbei die Genauigkeit der Modellierung zu prüfen. Eine mögliche Modellierung solcher Unterbrechungen mit einer Verhaltensabbildung von der Schalterebene auf der Logikebene wird im Abschnitt 4.2.2 demonstriert.

#### 4.3.2.4 Verzögerungsfehlermodell

Kein elektronisches Element ist in der Lage seine Funktion in unendlich kurzer Zeit auszuführen. Die entstehenden Verzögerungen sind eine wesentliche Restriktion für den maximal zu verwendenden Schaltungstakt. Bedingt durch Defekte, wie erhöhte Leitwiderstände oder falsche Diffusionen im Bereich der Transistoren [7], kann eine Verzögerung entstehen, die einen Betrieb der Schaltung innerhalb der Spezifikation nicht mehr erlaubt. Ein solcher Sachverhalt lässt sich mit dem Verzögerungsfehlermodell beschreiben. Da eine Schaltung im Ruhezustand keinen Fehler zeigt und dieser Fehler erst in Abhängigkeit des Schaltungstaktes erscheint, gehört er zu den dynamischen Fehlern. Bei einer schnellen Taktung kann ein Verzögerungsfehler bewirken, dass sich in der Schaltung Werte nicht ändern. Es scheint, als wenn sie eine unendliche Verzögerung besäßen. Diese Fehler sind Übergangsfehler und stellen einen Sonderfall dar. Das Verhalten kommt einem vorübergehenden Haftfehlermodell recht nah. Aus diesem Grund sind Testmuster, die aus Haftfehlermodellierungen stammen, oftmals geeignet, auch Übergangsfehler aufzudecken [157].

Das Listing 4.13 zeigt einen einfachen Verzögerungsfehler für ein AND-Gatter.

---

```

void and2::doit() {
2   if (fi.read()) {
       next_trigger(2,SC_NS);
4       Z0.write(A0.read() & A1.read());
       }
6   else {
       Z0.write(A0.read() & A1.read());
8   }
}

```

---

Listing 4.13: Einfacher Verzögerungsfehler für ein AND-Gatter

In Abhängigkeit eines Eingangs *fi* zur Fehlerauslösung (Zeile 2) wird eine sofortige AND-Verknüpfung oder eine verzögerte realisiert. Die Verzögerung ist für die steigende und fallende Ausgabe gleich lang.

Während bei früheren Halbleiterschaltungen hauptsächlich die Verzögerungen von Gattern näher betrachtet wurden, ist heutzutage in schnellen Schaltungen der Einfluss der Verbindungen ebenso von Bedeutung. Um eine Verzögerung auf einer Signalleitung zu erreichen, wird äquivalent wie beim Gatter-Modell vorgegangen. Das Verzögerungsmodell kann man hierzu in einem Saboteur-Modul implementieren (→5.4.1).

Mitunter gibt es Defekte, die nur Verzögerungsfehler für eine Flanke erzeugen [7]. Außerdem existieren Signalübergänge, die recht langsam erfolgen. Zwischen den definierten logische Werten '1' und '0' wird über einen längeren Zeitraum ein undefinierter Pegel eingenommen. Solche Eigenschaften bezeichnet man als *slow to rise* für stark verzögerte Anstiege und *slow to fall* für stark verzögerte fallende Flanken. Die Abbildung 4.19 verdeutlicht dies an einem AND-Gatter.

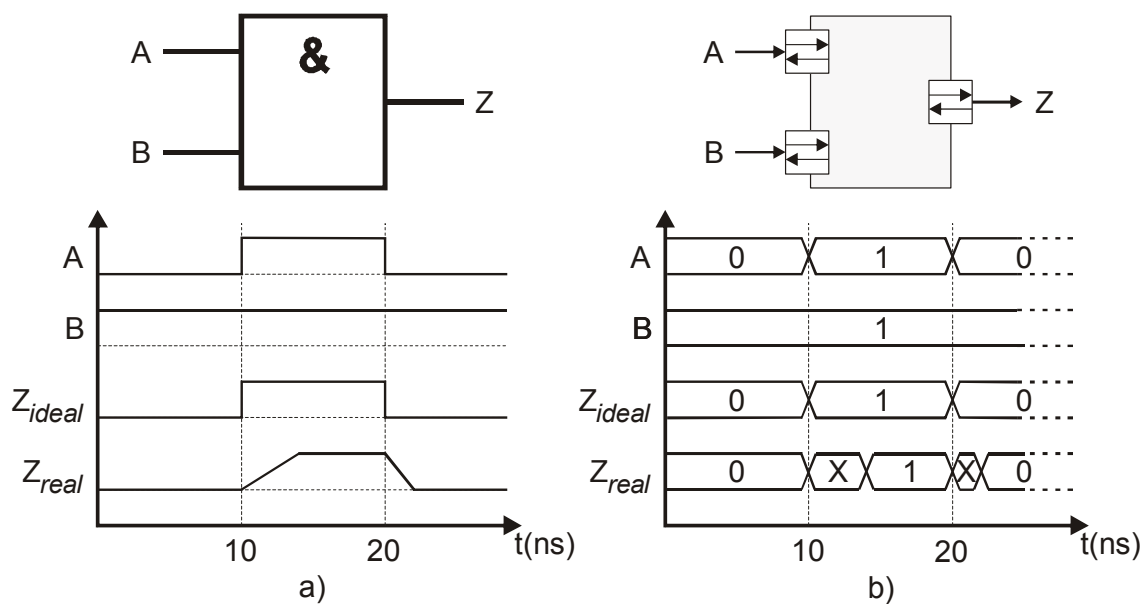


Abbildung 4.19: AND-Gatter mit a) realem und b) logischem Verzögerungsverhalten

Bei einem Übergang auf logisch '1' ist in a) eine Verzögerung von 4 ns angegeben und die fallende Flanke benötigt 2 ns im Beispiel. Bei der logischen Implementierung b) wird der undefinierte Bereich des steigenden und fallenden Pegels an  $Z_{real}$  durch das unbekannte 'X' charakterisiert. Das Listing 4.14 zeigt für das Beispiel eine Implementierung in SystemC.

---

```
SC_MODULE (and2) {
2   sc_out<sc_logic> Z;
   sc_in<sc_logic> A, B;
4   sc_logic temp_z;
   bool vd;
6
   void doit();
8
   SC_CTOR (and2) {
10      SC_METHOD (doit);
        sensitive << A << B;
12      vd = true;
   }
14 };
```

---

```
void and2::doit() {
16   temp_z = A.read() & B.read();
   sc_time tdel(1, SC_NS);
18   sc_time tdel2;//(1, SC_NS);
   if (temp_z == sc_logic_1) tdel2 = tdel * 4;
20   else tdel2 = tdel * 2;
   if (vd) {
22      if (temp_z != Z.read()) Z.write(sc_logic_X);
        next_trigger(tdel2);
24      vd = false;
   }
26   else {
        Z.write(temp_z);
28      vd = true;
   }
30 }
```

---

Listing 4.14: Detailliertes Verzögerungsmodell des AND-Gatters

Da der Umgang mit der Funktion *next\_trigger()* zur nötigen Modellierung der dynamischen Sensitivität nicht ganz trivial ist, sieht die Implementierung ein wenig umständlich aus. Aus diesem Grund folgt eine kurze Erläuterung. Das AND-Gatter wird ganz normal durch eine Änderung am Eingang aktiviert (Zeile 11). Bei einem ersten Aufruf erfolgt zunächst im Falle einer Ausgangsänderung die Ausgabe des logischen 'X' (Zeile 22). In Abhängigkeit des später anzunehmenden Pegels dauert sie 4 ns (Zeile 19) oder 2 ns (Zeile 20). Der nächste Funktionsaufruf wird nicht durch eine Eingangsänderung, sondern durch die *next\_trigger*-Funktion hervorgerufen.



Damit nicht wieder eine *next\_trigger*-Ausführung (Zeile 23) geschieht, wird das Flag *vd* verwendet, so dass beim zweiten Funktionsaufruf die Ausgabe der AND-Verknüpfung erfolgt. Auf die Darstellung des Umschaltens zwischen fehlerfreier und fehlerhafter Komponente wurde der Einfachheit halber verzichtet.

#### 4.3.2.5 Temporäre Fehler

Temporäre Fehler beruhen nicht auf physikalischen Defekten, sondern werden beispielsweise von außerhalb provoziert. Daher spielen solche Fehler für die klassische Fehlersimulation zur Erzeugung von Testmustern zur Aufdeckung defektbezogener Fehler für den Produktionstest keine Rolle. Sie sind jedoch wichtig bei der Untersuchung von Schaltungen bei der simulierten Fehlerinjektion (→5.2.1.3).

Da in SystemC keine nähere Betrachtung der Signalverläufe in Form wert- und zeitkontinuierlicher Eigenschaften erfolgt, sondern in Form von logischen Datentypen, ähneln die Fehlermodelle für temporäre Fehler denen der permanenten Fehler bezüglich der verwendeten Werte. Die Besonderheit für temporäre Fehler steckt in ihrer vorübergehenden Erscheinung. Ebenso ist eine Unterscheidung bei der Fehlermodellierung bezüglich intermittierender oder transienter Art kaum relevant, da nicht die Ursache sondern die Auswirkung modelliert werden soll. Diese sind bei diesen Fehlertypen ähnlich (→4.3.1) [174]. Temporäre Fehler können bezüglich ihrer Wirkung in drei Klassen eingeteilt werden [151]:

- Ein Fehler wird wirksam und führt zu einem Fehlerzustand (→5.1.2).
- Ein Fehler wird durch eine Folgeoperation überschrieben und hat somit keinen Effekt.
- Ein Fehler bleibt verborgen (latent) und wird nicht wirksam.

Eine einfache Fehlermodellierung sieht so aus, dass ähnlich wie bei den Haftfehlern eine logische '1' oder '0' bzw. wie bei den Brückenfehlern ein 'X' injiziert werden und nach einer gewissen Zeitspanne die Fehlerinjektion wieder zur Aufhebung kommt. Dieses Vorgehen ist geeignet, um Fehler zu modellieren, die mindestens einen Taktzyklus wahren. Näheres zu den Methoden der Fehlerinjektion ist ausführlich im Kapitel 5 beschrieben.

Eine anderes Fehlermodell für transiente Fehler ist das *Bit-Flip*-Modell. Dieses Modell repräsentiert den Wechsel eines Signalwertes auf einer Leitung, in einem Register oder einer Speicherzelle. Bei diesem Modell sind die beiden Arten *flip-to-0* und *flip-to-1* üblich. Das Listing 4.15 zeigt eine Implementierung eines Bit-Flip-Fehlermodells in SystemC. Dieses ist so ausgeführt, dass es bei einer Aktivierung einen komplementären Wechsel des aktuellen Zustands bewirkt (Zeile 4). Bei einer vorhandenen logischen '1' entspricht es einem *bit-flip-0* und bei einer logischen '0' einem *bit-flip-1*.

```

2   int v_fi = fi.read();
   switch (v_fi) {
4       . . .
       case 4: Z.write(~(A.read())); //bit-flip
6           break;
       default: Z.write(A.read()); //normale Durchschaltung
8           break;
   . . .

```

Listing 4.15: Bit-Flip-Fehlermodell in SystemC

Die bisher beschriebenen temporären Fehlermodelle gelten für mindestens einen Taktzyklus. Ist es jedoch nötig, Fehler mit einer Zeitdauer von weniger als einem Takt zu injizieren, so sind diese Fehlermodelle unbrauchbar. Solche Fehlermodellierungen werden aber zunehmend wichtiger, um z. B. Erscheinungen in elektronischen Schaltungen des *deep-submicron*-Bereichs und das Verhalten dort vorhandener Fehlerkorrekturmaßnahmen zu untersuchen [173].

Da SystemC die Modellierung von Multi-Takt-Systemen gestattet, bietet es sich an, diese Eigenschaft zu nutzen, um kurze Fehlerinjektionen zu realisieren.

Die Abbildung 4.20 zeigt a) das zeitliche Schema und b) die Definition der Mehrfach-Takte in SystemC.

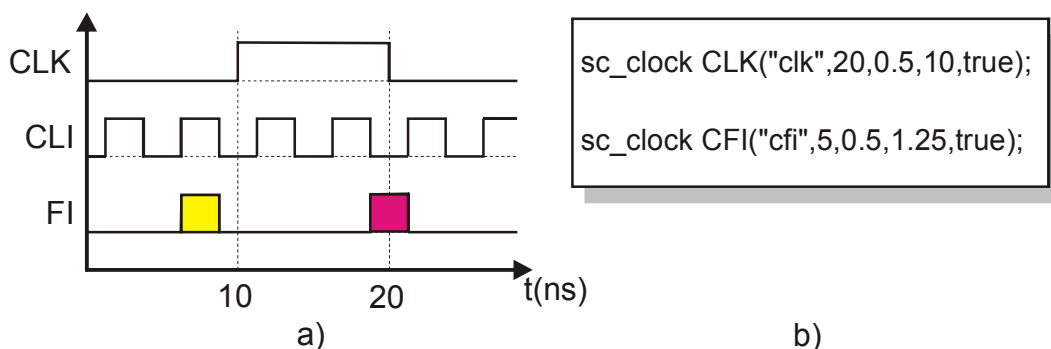


Abbildung 4.20: Mehrfachtake für kurze transiente Fehler

Der Takt *CLK* dient als normaler Schaltungstakt und das Taktsignal *CFI* ist ein Hilfsmittel zur Fehlerauslösung. Der Wert *FI* zeigt die Dauer des Fehlers in der Abbildung. Der erste aufgeführte Wert wird für eine Schaltung mit flankengetriggerten Flip-Flops nicht effektiv in Erscheinung treten, im Gegensatz zum zweiten, da er nicht während einer Taktflanke gültig ist.

Die Abbildung 4.21 zeigt schematisch die Umsetzung der Fehlermodellierung kurzer transienter Fehler in ein SystemC-Modell. Auf der linken Seite wird ein Injektionsmanager (IM) mit dem schnelleren Fehlertakt *CFI* angesteuert als das DUT, welches den normalen Schaltungstakt *CLK* bekommt. Der IM kümmert sich um die zeitliche Abfolge der Fehlerinjektionen.

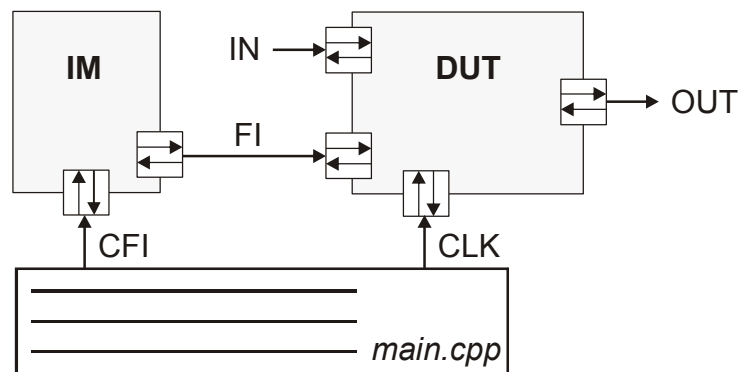


Abbildung 4.21: SystemC-Modell für kurze transiente Fehler

Das in der Abbildung 4.21 dargestellte Schema ist geeignet, um Fehlerinjektionen in Saboteuren (→5.4.1) und Mutanten (→5.4.2) vorzunehmen. Für die Fehlerinjektionstechnik der Simulations-Kommandos (→5.4.3) wird der Takt feiner modelliert (→Listing 5.11). Die gleiche Vorgehensweise ist dort für den Fehlertakt *CFI* möglich. Dieser wird dann in den Schaltungstakt *CLK* „geschachtelt“. Ein Vorteil dieser Implementierung liegt darin, dass der Injektionsmanager entfällt.



# 5 Fehlerinjektion in digitale Hardwaremodelle

Für heutige Computersysteme entstehen zunehmende Herausforderungen, wenn es darum geht, ihre Arbeit zuverlässig auszuführen. Bei Consumer-Anwendungen mag das Auftreten eines Fehlers nur störend sein, aber in sicherheitskritischen Steuerungen, wie sie bei der Verkehrstechnik, Medizintechnik, Raumfahrttechnik oder in Atomkraftwerken vorkommen, kann ein unerwünscht abweichendes Verhalten ernsthafte Konsequenzen nach sich ziehen. Hier führen fehlerhafte Vorgänge oder Ausfälle immer wieder zu hohen materiellen Schäden, wie beispielsweise dem Fehlstart einer Raumfahrttrakte oder zum Verlust von Menschenleben, z. B. durch einen Flugzeugabsturz. Aber nicht nur in sicherheitskritischen Anwendungen hat die Verlässlichkeit einen hohen Stellenwert. Zunehmend kleinere Strukturen in der Halbleitertechnologie und höhere Packungsdichten machen auch Low-cost High-performance Prozessoren künftig anfällig gegenüber Fehlererscheinungen. Ohne zusätzliche Maßnahmen wird hier auch der reguläre Betrieb gefährdet sein [93,94]. Um das gewünschte Verhalten für ein System trotzdem zu gewährleisten, bekommt es zusätzliche Eigenschaften der Fehlertoleranz [151]. Solche Maßnahmen erreicht man vorwiegend durch den Einsatz von Redundanzen oder Kodierungen. Je nach Art des Fehlers, ob permanent oder temporär, sind unterschiedliche Strategien der Korrektur erforderlich. Während bei zeitweiligen Fehlern das nochmalige Ausführen einer Operation genügt, kann es bei einem festen Fehler notwendig sein, zusätzlich eine Selbstreparatur durchzuführen.

Die Verlässlichkeit eines Systems muss natürlich auch validiert werden, z. B. um die korrekte Implementierung von Fehlertoleranzmechanismen nachzuweisen. Seit Jahren ist hierzu die Fehlerinjektion ein effektives Mittel zur Bewertung der Zuverlässigkeit. Dabei werden in ein laufendes System gezielt Fehler injiziert und die darauffolgenden Reaktionen untersucht.

Im ersten Teil des Kapitels sind einige Definitionen aufgeführt, um den Kontext der Fehlerinjektion besser zu verstehen. Außerdem werden Fehlereffekte und die grundsätzliche Vorgehensweise einer Fehlerinjektion näher beleuchtet. Teil 2 gibt einen Überblick zu den bisher bekannten Fehlerinjektionsgruppen und Teil 3 geht speziell auf die simulierte Technik von HDL-Beschreibungen ein mit einem Bezug auf den aktuellen Stand der Technik bei SystemC-Modellen. Der verbleibende Teil des Kapitels beschäftigt sich mit eigenen Implementierungen von Fehlerinjektionen und Experimenten.

## 5.1 Verlässlichkeit und Fehlereffekte

### 5.1.1 Verlässlichkeit

Um ein verlässliches System zu entwickeln, genügt es nicht, ausschließlich auf einen sorgfältigen Entwurf zu achten. Es werden definierte Mechanismen benötigt, um die Verlässlichkeitskriterien zu erfüllen. Drei Punkte sind hierfür wesentlich [123]:

1. Spezifikation der Anforderungen für die Systemverlässlichkeit,
2. Entwurf und Umsetzung des Systems, so dass die geforderte Verlässlichkeit gegeben ist,
3. Validierung des Systems.

Um Verlässlichkeit für ein System zu erreichen bzw. zu bestimmen, existieren diverse Mittel (engl. *means*):

- **Fehlervermeidung** (*fault prevention*) - z. B. durch einen qualitativ hohen Entwurfsansatz,
- **Fehlertoleranz** (*fault tolerance*) - z. B. durch den Einsatz von Redundanzen,
- **Fehlerbeseitigung** (*fault removal*) - das Aufspüren und Entfernen von Entwurfsfehlern,
- **Fehlervorhersage** (*fault prediction*) - anhand von Fehlerinjektionen, z. B. auf experimentellem Weg, kann eine statistische Aussage über die sichere Funktionsweise eines Systems getroffen werden.

Die Verlässlichkeit (*dependability*) eines Systems, zuweilen auch missverständlich als Zuverlässigkeit in diesem Kontext bezeichnet, lässt sich nicht durch ein einzelnes Maß ausdrücken. Sie wird vielmehr durch eine Vielzahl von Metriken bestimmt [123].

Einige wichtige Kennzahlen [127,2] sind:

- **Zuverlässigkeit** (*reliability*) ist eine bedingte Wahrscheinlichkeit, dass das System innerhalb eines Intervalls  $[t_0, t]$  korrekt arbeitet, wenn es zum Zeitpunkt  $t_0$  korrekt funktionierte [128]. Dies betrifft den Verlauf eines Dienstes.
- **Verfügbarkeit** (*availability*) ist die Wahrscheinlichkeit, dass ein System korrekt arbeitet und verfügbar ist zu einem Zeitpunkt  $t$ . Dies charakterisiert die Bereitschaft zum Gebrauch.
- **Sicherheit** oder **Angriffstoleranz** (*safety*) ist die Wahrscheinlichkeit, dass das System beim Auftreten eines Problems weiterhin korrekt arbeitet oder seinen Betrieb so fortsetzt, dass andere laufende Systeme nicht gestört werden.

Weiterhin dürfen Personen im Zusammenhang mit dem System nicht gefährdet werden. Dies charakterisiert das Nichtauftreten von katastrophalen Konsequenzen für die Umgebung.

- **Vertraulichkeit** (*confidentiality*) oder **Integrität** meint die Eigenschaft, dass vertrauliche Daten geheim bleiben und dass die Authentizität der Kommunikation garantiert ist.
- **Wartbarkeit** (*maintainability*) ist die Wahrscheinlichkeit, dass ein ausgefallenes System innerhalb eines bestimmten Zeitbereichs wieder repariert werden kann.

Neben den eben genannten, wohl wichtigsten Metriken gibt es noch einige weitere. Dies sind z. B. die *Mean Time To Failure* (MTTF), welche die zu erwartende Zeit bis zum Auftreten eines ersten Ausfalls bezeichnet und die *Mean Time Between Failure* (MTBF), die die Zeitspanne zwischen zwei Ausfällen meint.

Die Verlässlichkeitsmetriken wurden oft über einen längeren Einsatzzeitraum ermittelt. Um eine aussagekräftige statistische Aussage zum Auftreten von Ausfällen zu bekommen, ist dieses Vorgehen schon aus Zeitgründen für die meisten Anwendungen unpraktisch. Ein Ausweg besteht hier durch die Verwendung von Fehlerinjektionstechniken.

### 5.1.2 Das Auftreten von Fehlern

Nicht immer erfüllt ein System die Funktion, für die es ausgelegt ist. Die Gründe und Konsequenzen der inkorrekten Abweichung bezeichnet man auch als Faktoren der Verlässlichkeit. In der Abbildung 5.1 sind die entsprechenden Zusammenhänge in Anlehnung an [151,113] dargestellt. Ursache eines Ausfalls ist ein Fehler (*fault*). Dieser kann beispielsweise ein physikalischer Defekt oder auch eine vorübergehende Störung sein, welche durch elektromagnetische Störung ausgelöst wurde. Das Auftreten muss allerdings nicht zwangsweise zu einer Abweichung der Systemfunktion führen. Nicht alle Fehlererscheinungen werden auch wirksam. Ist jedoch ein Fehlereffekt (*fault*) entstanden, so kann es einige Zeit dauern (*fault latency*), bis er sich auch als Fehlerzustand bzw. Irrtum (*error*) manifestiert. Das Auslösen eines Fehlerzustandes ist eine notwendige Voraussetzung für einen Ausfall (*failure*). Allerdings führt nicht jeder Fehlerzustand zu einem Ausfall. Ein unerwünscht gekipptes Bit in einem Register kann zum Beispiel in einem nächsten Schritt durch eine korrekt arbeitende Operation überschrieben werden, bevor eine andere Operation aus diesem Register liest. Ist der Fehlerzustand aktiv, so dauert es einen gewissen Zeitraum (engl. *error latency*) bis ihm ein Ausfall folgt. In dieser Latenzzeit muss der Fehlerzustand erkannt werden (*error detection*) und es müssen fehlerkorrigierende Maßnahmen wirken, wenn es nicht zu einem Ausfall kommen soll.

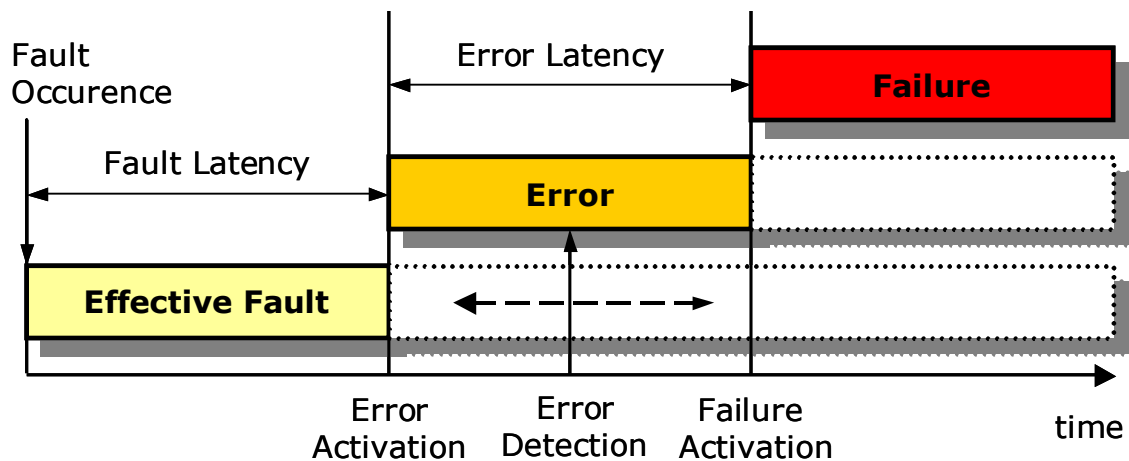


Abbildung 5.1: Zeitlicher Zusammenhang zwischen Fault, Error, Failure

### 5.1.3 Der Fehlerraum

Das Auftreten eines Fehlers lässt sich anschaulich mit dem Fehlerraum I (engl. *fault space*) darstellen [123]. Dieser ist ein mehrdimensionaler Raum, welcher:

- die Zeit des Auftretens und die Dauer des Fehlers,
- den Typ oder die Art des Fehlers und
- den Fehlerort vereint.

In Abhängigkeit von der Komplexität des Systems und der verwendeten Fehlermodelle kann dieser Fehlerraum recht groß werden. Für eine genaue Abbildung eines real auftretenden Fehlers ist dazu die Verwendung geeigneter Fehlermodelle erforderlich (→4.3.2).

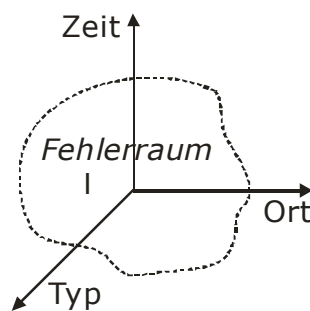


Abbildung 5.2: Fehlerraum



### 5.1.4 Das FARM-Set

1990 wurde eine Methode der physikalischen Fehlerinjektion zur Verlässlichkeitsvalidierung vorgestellt [130]. Hierzu wurden unter anderem theoretische Grundlagen der Fehlerinjektion vorgestellt. Das dort präsentierte FARM-Set, welches zunächst nur für die physikalische HW-Fehlerinjektion gedacht war, erwies sich später auch für die Betrachtung anderer Injektionsarten als grundlegend und nützlich.

Die Fehlerinjektion wird hierbei als eine Sammlung von Mengen betrachtet, den sogenannten FARM-Mengen (*sets*).

- Die Menge  $F$  (*faults*) beschreibt im Allgemeinen die Menge der Fehler und gehört zur Eingabe-Domäne. Abhängig vom Abstraktionslevel der Fehlerinjektion ist sie ein stochastischer Prozess oder besteht aus sehr konkreten Fehlern.
- Die Menge  $A$  (*activations*) bezeichnet eine Menge von Aktivitäten. Das sind normalerweise die Eingabedaten, die auch bei einer normalen Funktion verwendet werden. Für den Fertigungstest können dies auch die Eingabetestmuster sein oder bei einem Mikroprozessor das zu ladende Programm. Die Mengen  $F$  und  $A$  sind für das Erzeugen eines Fehlerzustandes verantwortlich.
- Die Antworten eines Systems bilden die Menge  $R$  (*readouts*). Die Menge  $R$  gehört zum Ausgabebereich der Fehlerinjektionen.
- Die Menge  $M$  (*measures*) bezeichnet die Messungen und Auswertungen aus den anderen Mengen bezüglich der Verlässlichkeit. So kann die Menge  $M$  beispielsweise eine übliche Verlässlichkeitsmetrik wie die Verfügbarkeit enthalten.

In der Abbildung 5.3 sind die FARM-Mengen der Fehlerinjektion dargestellt.

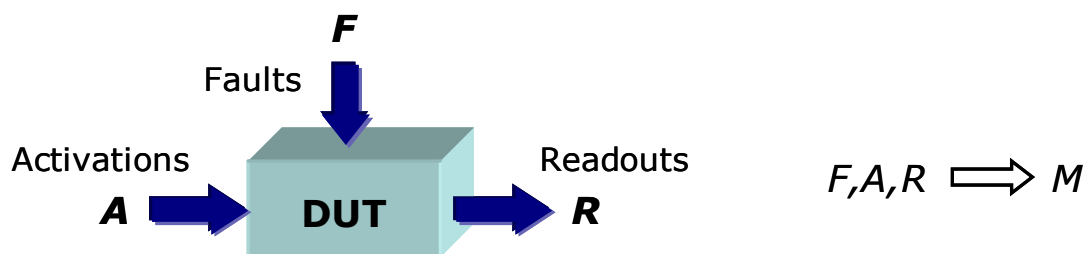


Abbildung 5.3: FARM-Mengen

Die FARM-Mengen stellen die Grundlage für die meisten Fehlerinjektionsexperimente dar. Ausgehend von diesen findet man zuweilen Erweiterungen. Zum Beispiel werden in [133] zusätzliche Aussagen P (Predicate) explizit hinzugefügt, welche dort speziell der Auswertung der Fehlerinjektion dienen.

## 5.2 Fehlerinjektionen

Erste Fehlerinjektionsexperimente unternahm man in den 70er Jahren um die Verlässlichkeit von fehlertoleranten Computern abzuschätzen [113]. Einige Zeit später war die Fehlerinjektion fast das ausschließliche Instrument der Industrie, um einige Verlässlichkeitsparameter von hoch verfügbaren Systemen zu bestimmen. Ab Mitte der 80er Jahre begannen akademische Experimente mit der Fehlerinjektion. Dabei ging es zunächst um das Verständnis der Fehler-Propagation und der Untersuchung neuer Mechanismen zur Fehlererkennung. Seitdem hat sich das Anwendungsfeld der Fehlerinjektion stark erweitert, insbesondere dient sie zur Charakterisierung der Verlässlichkeit von Systemen auf unterschiedlichen Abstraktionsebenen.

### 5.2.1 Gruppen der Fehlerinjektion

Fehlerinjektionen können auf verschiedene Weise in Hardware vorgenommen werden. Bei bisherigen Publikationen haben sich hier drei Haupttechniken herausgebildet [123]:

- Hardware Fault Injection (HWFI),
- Software Fault Injection (SWFI) und
- Simulated Fault Injection (SFI).

Neben diesen drei Verfahren gibt es noch Mischformen, die als Hybrid Fault Injections bezeichnet werden [134].

Die verschiedenen Fehlerinjektionen können anhand ihrer Eigenschaften [135] charakterisiert werden. Typische Eigenschaften der Fehlerinjektion sind:

- Die **Erreichbarkeit** (*reachability*) drückt die Fähigkeit aus, einen Fehlerort zu erreichen.
- Die **Steuerbarkeit** (*controllability*) bezeichnet die Fähigkeit, zu bestimmen, wann und wo eine Fehlerinjektion stattfindet.
- Die **Wiederholbarkeit** (*repeatability*) ist die Fähigkeit ein Fehlerinjektionsexperiment genau wiederholen zu können.
- Die **Reproduzierbarkeit** (*reproducibility*) bezieht sich hingegen auf das Reproduzieren von statistischen Ergebnissen von Experimenten.

- Die **Aufdringlichkeit** (*intrusiveness*) bezieht sich auf das Niveau der unerwünschten Beeinflussung der Fehlerinjektion im Verhalten des Zielsystems. Diese kann hinsichtlich Raum und Zeit unterschieden werden.
- Die **Flexibilität** (*flexibility*) sagt aus, wie einfach sich ein Injektionsziel ändern lässt.
- Die **Effektivität** (*effectiveness*) drückt das Vermögen aus, wie verschiedene Fehlermechanismen aktiviert und untersucht werden können.
- Die **Effizienz** (*efficiency*) bezieht sich auf den Aufwand der durchzuführenden Fehlerinjektion und der benötigten Zeit.
- Die **Beobachtbarkeit** (*observability*) bezeichnet die Fähigkeiten des Bemerkens, des Messens und der Möglichkeit des Aufzeichnens von Fehlereffekten.

### 5.2.1.1 Hardware Fault Injection

Bei der Hardware Fault Injection (HWFI) oder auch Hardware-basierten Fehlerinjektion [112,135] werden die Fehler auf der physikalischen Ebene induziert. Hier werden Fehler in einem Systemprototyp über seine eigene Hardware injiziert. Diese können zu Änderungen von physikalischen oder elektrischen Eigenschaften führen. Das Verfahren geht gelegentlich mit der Zerstörung des Zielsystems einher.

Eine Variante der Fehlerinjektion in Hardware ist die kontaktbehaftete Injektion. Eine verbreitete Methode ist die Störeinbringung an den Pins der Chips, zum Beispiel durch einen bestimmten Pegel [136,130]. Diese wird deshalb auch als *pin-level fault injection* bezeichnet. Außerdem sind mit diesem Weg auch Überbrückungen (engl. *bridging faults*) möglich. Ohne Problem realisiert man auf diese Weise permanente und zeitweise Fehlerinjektionen. Die Schwierigkeit besteht beim *pin-level*-Verfahren und bei hochintegrierten Schaltkreisen darin, dass interne Reaktionen nach Fehlereinflüssen nur unzureichend wiedergegeben werden.

Eine andere Möglichkeit ist die strahlenbasierte Injektion, welche energiereiche Partikel zuführt. Diese bewirkt SEUs (*single event upsets*), welche sich in Bit-Flip-Fehlern äußern. Solche Injektionen können elektromagnetischer Natur [131], stark ionische Strahlung [137] oder Laserquellen [139] sein.

Eine andere Variante ist die *Scan-Chain Implemented Fault Injection* (SCIFI), wo die Fehlerinjektion über Scanketten erfolgt, z. B. in Konformität zum IEEE 1194.1 Standard [192]. Diese Technik erlaubt eine bessere Einstellbarkeit, Beobachtung und Kontrolle als andere HWFI-Verfahren. Mit zusätzlich eingebauter Logik könnte diese Variante allerdings noch weiter verbessert werden.

Daneben kann man noch eine HWFI-Fehlerinjektion verursachen, indem man Störungen auf der Versorgungsspannung erzeugt [138].

Das HWFI-Verfahren setzt typischerweise ein fertiges Zielsystem voraus und benötigt noch zusätzliche Hardware, die zum Teil sehr aufwändig ist.

Außerdem lassen sich nur wenige Aussagen zum inneren Verhalten von Komponenten treffen, so dass Analysefähigkeiten für Verbesserungsvorschläge schlecht möglich sind. In der Designphase ist die HWFI nicht anwendbar.

### 5.2.1.2 Software Fault Injection

Die Software Fault Injection (SWFI) [112,135] findet ebenso wie die HWFI im physikalischen System statt, z. B. in einem Prototypen. Vorteile dieser Technik liegen darin, ohne zusätzliche Hardware auszukommen und flexibler zu sein als HWFI-Verfahren. Die Fehlerinjektion geht so vor, dass zusätzlicher Code in die Software eingefügt oder bestehender manipuliert wird. Mit der SWFI werden Hardware- und Softwarefehler nachgebildet. Durch das Setzen eines Registerinhaltes kann z. B. ein vorübergehender stuck-at-1 Fehler erzeugt werden.

Bei den SWFI-Techniken unterscheidet man zwei Kategorien, zum einen die *pre-runtime injection* und zum anderen die *runtime injection*. Bei der pre-runtime injection oder auch compile-time injection werden die Fehler schon vor dem Kompilieren hinzugefügt. Da die Fehler fest eingebunden sind, kann man sie während der Laufzeit nicht mehr ändern. Daher eignet sich dieses Verfahren gut zur Nachbildung permanenter Fehler. Weil hier Haltepunkte und Verzweigungen entfallen, besitzt sie Geschwindigkeitsvorteile bezüglich der Ausführung.

Bei der runtime injection werden die Fehler auch vor dem Laden in das Zielsystem im Code abgelegt. Jedoch lassen sich die Fehler während der Laufzeit durch den Gebrauch von Haltepunkten und Verzweigungen zu- und abschalten. Der Nachteil liegt in einem zusätzlichen Simulationszeit-Overhead.

Nachteilig an der SWFI-Methode ist, dass nicht alle Fehlerorte erreicht werden. Während das auszuführende Programm sich noch gut instrumentieren lässt, sind Hardwarefehler bedingt über die mit der Software erreichbaren Komponenten wie Speicher und CPU-Register möglich. Ebenso schwierig gestaltet sich aus der Softwaresicht die Fehlerinjektion in das Betriebssystem. Ein weiterer Nachteil besteht darin, dass die SWFI nicht während der Entwurfsphase durchgeführt werden kann.

### 5.2.1.3 Simulated Fault Injection

Bei der Simulated Fault Injection (SFI) bzw. simulierten Fehlerinjektion formt man aus der Hardware und der Software des Zielsystems ein Modell. Dieses Modell wird anschließend einem Simulator zugeführt. Prinzipiell eignen sich für die Simulation, abhängig von der gewählten Abstraktionsebene, diverse Programme. Auf der algorithmischen Ebene kann dies Matlab [142] sein und auf der elektrischen Ebene findet man z. B. SPICE-Simulatoren [141]. Besonders verbreitet sind Fehlerinjektionsexperimente in Hardwarebeschreibungssprachen (→5.3).

Der Vorteil dieser Methode liegt darin, dass man keine reale Hardware des zu untersuchenden Objektes benötigt. Außerdem ist die Erreichbarkeit, Steuerbarkeit und Beobachtbarkeit sehr gut. Zudem lässt sich dieses Verfahren schon in der Designphase und auf verschiedenen Abstraktionsebenen einsetzen (→2.1). Ebenso bietet sie meist die Möglichkeit, sowohl transiente als auch permanente Fehler zu injizieren. Nachteilig ist bei der SFI die oft deutlich höhere Simulationszeit im Vergleich zu den physikalischen Fehlerinjektionen.

In [145] und [135] sind Vergleiche von vielen Injektionsverfahren (HWFI, SWFI, SFI) aufgeführt. Es wird dabei besonders deutlich, dass nahezu alle vorgestellten Anwendungen bezüglich der Modellierungsebene des Entwurfs und hinsichtlich der Fehlerinjektionseigenschaften recht beschränkt sind.

## 5.3 Simulierte Fehlerinjektion in HDL-Beschreibungen

Die Technik der simulationsbasierten Fehlerinjektion in Hardwarebeschreibungssprachen ist Gegenstand zahlreicher Arbeiten. Eine andere Möglichkeit der Verwendung einer HDL für die Fehlerinjektion ist die physikalische Variante. So kann man auch eine Modellbeschreibung mit zusätzlichen Fehlerinjektionen instrumentieren und dieses neue Modell direkt in Hardware synthetisieren. So ließe sie sich z. B. auf einem FPGA ausführen. Eine solche kontrollierbare HW-basierte Fehlerinjektion wird in [143] vorgeschlagen.

Zur simulierten Fehlerinjektion wurden besonders viele Arbeiten mit VHDL publiziert [144,132,124,126], was sicher daran liegt, dass diese Sprache Modellierungen auf verschiedenen Abstraktionsebenen erlaubt.

Die Abbildung 5.4 zeigt in Anlehnung an [124] eine Systematik der verschiedenen Fehlerinjektionen in HDLs nach dem heutigen Stand der Technik.

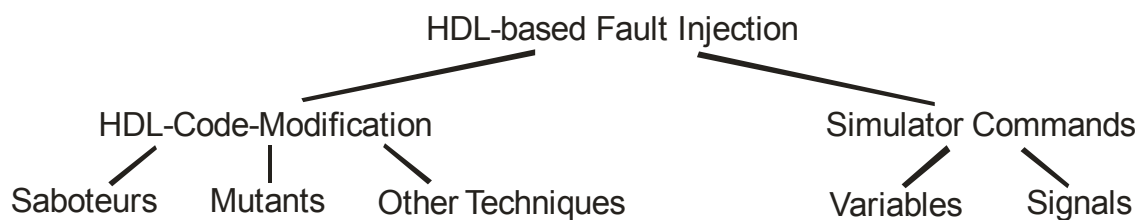


Abbildung 5.4: Eine Systematik von simulierten Fehlerinjektionen bei HDLs

Die gewählten Bezeichnungen haben sich hierbei als Quasi-Standard etabliert. Die Praktiken, welche den HDL-Code verändern, erzeugen ein syntaktisch korrektes aber im Verhalten fehlerhaftes Modell.

*Saboteurs* (dt. *Saboteure*) sind spezielle Komponenten, die dem Original-Modell hinzugefügt werden. In Hardwarebeschreibungen ändern diese Module die Werte oder das zeitliche Verhalten von Signalen. Eine Aktivierung eines Saboteurs führt zu einer Abweichung vom ursprünglichen Zustand und es entsteht so eine Fehlerinjektion. Die Saboteur-Methode eignet sich typischerweise nur für strukturelle Beschreibungen [125]. Bei der Saboteurmethode unterscheidet man grob in die serielle und die parallele Methode. Die serielle Methode wird im einfachsten Fall in den Signalweg zwischen zwei Komponenten eingefügt. Die parallele Methode wird zusätzlich an einen vorhandenen Signalweg angeschlossen. Es lassen sich so Busstörungen provozieren.

*Mutants* (dt. *Mutanten*) führen zu einer Veränderung der Schaltungsbeschreibung. Nach [148] kommt die Idee vom mutation testing, welches in [147] für das fehlerbasierte software testing vorgestellt wurde. Da Hardwarebeschreibungssprachen auch Programmiersprachen sind, bietet es sich an, diese bekannten textuellen Fehlermodelle aus dem Software-Test auch bei Hardwarebeschreibungssprachen einzusetzen. In [148,149] werden hierzu spezielle Mutant-Operatoren eingeführt, die z. B. arithmetische oder logische Operationen manipulieren oder Variablenzuweisungen ersetzen. In [152] werden hierzu acht Fehlerklassen definiert ( $\rightarrow$ A.2). Eine wichtige Besonderheit bei diesem Vorgehen ist, dass die Fehlerinjektion auf funktionaler Ebene stattfindet. Obwohl diese Methode mit Bitfehlerüberdeckung gut mit dem stuck-at-Fehlermodell korreliert, fehlt die realitätsnahe Fehlerinjektion für detaillierte Hardwarebeschreibungen. Außerdem kann hier aufgrund der verschiedenen Variationsmöglichkeiten und bei größeren Modellen der Ereignisraum sehr groß werden, was ohne Beschränkungen zu einer hohen Simulationszeit führt. Nach [150] ist dieses Verfahren eher dazu geeignet, Fehlerzustände (*errors*) als Fehler (*faults*) zu injizieren ( $\rightarrow$ 5.1.2). Statt der eben erwähnten Code-injizierenden Technik gibt es noch den Fall des Ersetzens von Programmblöcken. Dies bietet sich für Hardwarebeschreibungssprachen wie VHDL oder Verilog an, da dort Komponenten als kompakte Blöcke beschrieben sind. Zur Mutation wird die Originalkomponente gegen ein Modul mit der Erweiterung zur Fehlerinjektion ausgetauscht.

*Other Techniques* fasst die Methoden zusammen, die nicht in die oben beschriebenen Verfahren passen. So sind hierzu meist Arbeiten bekannt, die neue modifizierte Daten- oder Signaltypen definieren. Mit Hilfe dieser wird die Fehlerinjektion vorgenommen. In dem Fehlerinjektionstool VERIFY [146] werden z. B. zusätzliche Fehlerinjektionssignale (FIS) eingeführt, was zu einer konzeptionalen Erweiterung der HDL-Sprache führt. Diesen Signalen werden ein Intervallzyklus und eine Aktionsdauer zugewiesen. So lassen sich schnell Fehlerinjektionen realisieren, die nur lokal in einer Komponente definiert sind und für die kein globales Management während der Simulation notwendig ist.

In [129] wird ein spezieller Fehlerinjektionstyp für Busse eingeführt. Zusätzliche Maskierungs- und Kontrollvektoren erlauben dabei eine selektive Signaländerung auf Verbindungsleitungen.

Da bei diesen Techniken der neue Datentyp in die vorhandene Hardware-Beschreibung aufgenommen werden muss, gehören auch diese zu den modellverändernden Verfahren.

*Simulator Commands* (dt. *Simulatorkommandos*) werden auch manchmal als *Built-In Commands* [126] bezeichnet. Einige leistungsfähige HDL-Simulatoren (wie z. B. [24,25]) bieten eine Benutzungsschnittstelle, die es dem Anwender erlaubt, interaktiv mittels spezieller Kommandos den Simulationsvorgang zu steuern. Dabei existieren auch Befehle, die eine quasi-willkürliche Veränderung von Variableninhalten und Signalwerten gestatten. In Modelsim® [24], einem kommerziellen, weit verbreiteten HDL-Simulator, kann eine Fehlerinjektion für einen stuck-at-1-Fehler wie folgt aussehen:

```
force -freeze sim:/nand2/z0 1 0
```

In dem Beispiel wird mit dem Kommando `force -freeze` das Signal `z0` im Projektverzeichnis `sim` an der Komponente `nand2` nach einer Verzögerung von `0ns` auf den Wert `1` gezwungen.

In [144] wurde die Technik der Simulatorkommandos dazu benutzt, um in Modelsim® mit VHDL-Modellen eine Fehlersimulation auszuführen. Da der HDL-Simulator Script- und TCL-Programmierung erlaubt, konnte der Rechenvorgang gut automatisiert werden. Es zeigte sich, dass der zusätzliche Vorgang der Fehlerinjektion keine zusätzliche Simulationszeit erfordert. Das Resümee dessen war, dass die Simulation mit dieser Injektionsmethode sehr effizient verlief.

Ein wesentlicher Vorteil der Simulator Commands besteht darin, dass sie keine Modifikation der Schaltung erfordert und die Modelle deshalb nicht größer werden. Dies wirkt sich auch positiv auf die Simulationszeit aus. Von Nachteil ist jedoch, dass nur solche Fehlermodelle zu injizieren sind, für die entsprechende Spezialkommandos und Typen im Simulator existieren.

### 5.3.1 Bisherige Fehlerinjektionen mit SystemC

Bisher gibt es weitaus nicht so viele Publikationen zu Fehlerinjektionen in SystemC wie zu VHDL. Dies liegt zum einen sicher daran, dass SystemC im Vergleich zu anderen Hardware-Beschreibungssprachen noch nicht so lang verfügbar ist. Zum anderen an den grundsätzlichen Schwierigkeiten, die ein Wechsel von einer etablierten HDL zu einer neuen mit sich bringt.

In diesem Abschnitt werden drei interessante, publizierte Arbeiten vorgestellt, welche sich speziell mit SystemC-Beschreibungen befassen und auf Basis von Fehlerinjektionsmethoden auch prinzipiell eine Fehlersimulation ( $\rightarrow$ 3.2). gestatten. Für die vorgestellten Publikationen ist es typisch, dass sie hinsichtlich des Einsatzfalles bzw. der Fehlerinjektionstechnik spezialisiert sind und damit Einschränkungen unterliegen. Die am häufigsten verwendete Technik ist hierbei die Saboteurmethode.

Neben der Validierung eines Entwurfes per Simulation gibt es noch die Möglichkeit mittels Verifikation. Auf Basis von SystemC wurden hierzu in [44,45,46] einige Arbeiten vorgestellt.

### **5.3.1.1 “Error Simulation based on the SystemC design description language”**

In [114] wurde eine Fehlerzustandssimulation bzw. eine Error Simulation vorgestellt, die sich nicht mit der Fehlermodellierung auf abstrakten Beschreibungen beschränkt, sondern auch auf darunter liegenden Ebenen stattfinden kann. Augenmerk dieser Arbeit ist es, eine Umgebung zu haben, die der Auswertung von Fehlermodellen dient. Solche Betrachtungen sind beispielsweise wichtig, wenn die Zuverlässigkeit von komplexen Systemen anhand verschiedener Fehlermethoden betrachtet werden soll.

Diese Arbeit geht dabei von zwei Einschränkungen aus:

1. Die Fehlerzustandssimulation erfolgt auf Grundlage vorgegebener HDL-Primitive (typische Basiselemente der Gatterebene).
2. Der Ablauf erfolgt, ausgehend von der Spezifikation, so automatisch wie möglich.

Es wird in dieser Arbeit auf SystemC-Beschreibungen eingegangen, allerdings lassen sich die vorgestellten Prinzipien ebenso für VHDL oder Verilog nutzen.

Die Anwendung selbst kennt zwei Phasen:

1. In der ersten Phase wird der Code geparkt, in ein Zwischenformat abgelegt und mit zusätzlichem Code instrumentiert. Dieser zusätzliche Code ist ein Satz von zusätzlichen Ad-hoc-Funktionen. Diese Ad-hoc-Funktionen werden von einem Modul kontrolliert, welches sich außerhalb des DUT befindet. Mit diesem Kontrollmodul wird ein Fehlerzustand aktiviert oder deaktiviert.
2. Die zweite Phase erstellt automatisch eine Testbench. Hier wird sich Testbench-üblich um die Instanziierung der Module, das Anlegen von Eingabevektoren, der Aktivierung der Fehlerinjektionen und das Schreiben der Daten gekümmert. Zusätzlich erfolgt für jeden Fehler eine Analyse durch einen Vergleich mit der fehlerfreien Simulation. Die Testbench ist ebenfalls in SystemC geschrieben, so dass ein Standard-Kompiler ausreicht.



Hauptvorteil der vorgestellten Arbeit ist die Analysemöglichkeit und der Vergleich verschiedener Fehlermodelle, besonders im Hinblick auf die Untersuchung neuer, komplexerer Fehlermodelle. Es wird z. B. vorgeschlagen, dass man mit den neuen Fehlermodellen versuchen kann, Designfehler früh im Entwurfsprozess zu entdecken. Ein Vorteil beim Gebrauch von SystemC entsteht dadurch, dass der Designer sich nicht um den Simulator kümmern muss. Einzig das Fehlermodell muss neu erstellt werden, die Schaffung eines neuen Fehlersimulators ist unnötig. Es wurde außerdem festgestellt, dass der Overhead einer Fehlerinjektion gegenüber der fehlerfreien Simulation bei der Ausführung in allen Fällen zu vernachlässigen war.

Über Simulationsgeschwindigkeiten oder über die Größen der Experimentierschaltungen wird in dieser Arbeit nichts berichtet. Es fehlt ebenso eine Aufstellung der untersuchten Fehlermodelle. Nahezu zeitgleich wurde von dieser Forschungsgruppe eine Arbeit zur Error-Simulation in VHDL-Beschreibungen [117] publiziert. Dort verwendete man für verhaltensbeschreibende Modelle eine neue Injektionstechnik (Bedingungs-, Bitfehler), die zu einer neuen *Bit Coverage Metric* (Bit-Überdeckung) führte. Da diese verhältnismäßig gut mit dem Haftfehlermodell unterer Entwurfsebenen korreliert, konnten so Fehlerinjektionsuntersuchungen zwischen verschiedenen Entwurfsebenen unternommen werden. Mag die vorgestellte *Bit Coverage Metric* auch realitätsnäher sein, so entdeckt man mit ihr nur einen Teil wirklich auftretender Defekte.

Bei Bruschi et al. bzw. am Politecnico di Milano entstanden weitere Arbeiten zur Fehlersimulation [114,117]. Kennzeichnend an diesen Arbeiten ist, dass sie sich mit der Fehlersimulation von Finiten State Machines (FSM) befassen.

#### 5.3.1.2 “SystemC as a Complete Design and Validation Environment”

Eine Multi-Language-Umgebung mit der Möglichkeit zur Fehlersimulation wurde von Fin et al in [119] vorgestellt. Sehr ausführlich und mit deutlichem Fokus auf SystemC ist diese Validierungsumgebung auch in [38] zu finden. Sie dürfte derzeit die bekannteste Fehlersimulationsumgebung für SystemC sein, weshalb sie hier auch näher erläutert wird.

Diese Anwendung, auch als AMLETO bezeichnet, erlaubt sowohl die Eingabe von VHDL-Designs wie auch von SystemC-Beschreibungen. Zudem unterstützt sie die Untersuchung auf mehreren Ebenen des Entwurfs. Schaltungsbeschreibungen werden geparkt und in ein internes Format (IIR) gespeichert. Dieses interne Format kann wieder in eine VHDL- oder SystemC-Beschreibung überführt werden.

AMLETO selbst basiert auf SAVANT [120,121] und hat diese Anwendung erweitert.

SAVANT ist ein Public-Domain-Programm [123], welches hauptsächlich an der University of Cincinnati entstand. Diese Anwendung vollführt die automatische Übersetzung von VHDL-Beschreibungen in C++-Code. Wird die entstandene C++-Darstellung mit dem von der SAVANT-Library unterstützten Simulatorkernel verlinkt,

so entsteht ein C++-Programm, welches im Verhalten äquivalent zur ursprünglichen VHDL-Quelle ist. SAVANT arbeitet in zwei Phasen. In der ersten Phase wird der VHDL-Code in die *internal intermediate representation* (IIR) konvertiert. In der zweiten Phase entsteht der C++-Code. Eine Umwandlung zurück in das VHDL-Format ist ebenso möglich. Diese Eigenschaft nutzt AMLETO, um aus einer SystemC-Beschreibung VHDL-Code zu erzeugen. Nachteile bei SAVANT sind, dass der erzeugte Code sehr komplex ist und der Nutzer wenig Steuerungsmöglichkeiten bei den Simulationen hat. Zum Teil wird dies bedingt durch die Vermischung von Schaltungsbeschreibung und Simulationskernel. Dagegen wird bei SystemC stark zwischen Simulationskernel und Systembeschreibung unterschieden, das war auch ein wichtiger Grund für die Erweiterungen zu AMLETO.

Ziel von AMLETO war es, eine vollständige und effiziente Umgebung zur Validierung von VHDL- und SystemC-Beschreibungen zu liefern. AMLETO wurden hierfür folgende Module hinzugefügt:

- **SCRAM SystemC Analyzer:** Dieses Modul erlaubt die Konvertierung von SystemC-Beschreibungen in das interne IIR-Format.
- **IIR to SystemC Translator:** Diese Komponente erzeugt aus dem IIR-Format einen SystemC-Code. Eine sehr nützliche Eigenschaft liegt darin, dass bisherige VHDL-Modelle nun auch als SystemC-Beschreibungen vorliegen. So entstanden schnell neue Komponenten-Bibliotheken für das noch recht neue SystemC.
- **Control Module Generator:** Dieses Modul dient dem Test eines einzelnen Moduls, mehrerer Module oder der gesamten Architektur. Realisiert wird dies durch zusätzliche switches.
- **Error List Generator:** Dieser Teil analysiert die IIR-Darstellung und extrahiert aus ihr eine Error- bzw. Fehlerliste.
- **Error Injector:** Diese Komponente erzeugt anhand der Fehlerliste eine IIR-Beschreibung mit Fehlerinjektion.
- **TPG Generator:** Dies ist der Kern der Testumgebung. Es wird eine Architektur erzeugt, welche die Test-Pattern-Generierung, den switch-Manager und den Fehlerdetektor hinzufügt.

Die Abbildung 5.5 zeigt die AMLETO-Erweiterungen aus [119].

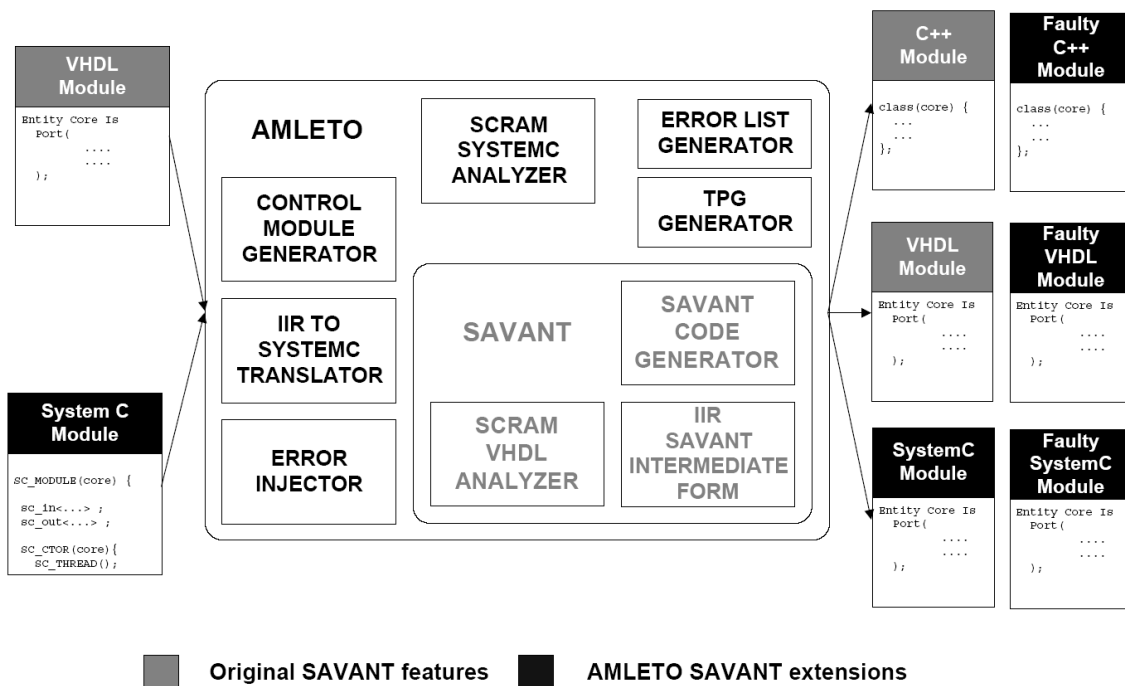


Abbildung 5.5: AMLETO-Architektur

Im Folgenden werden einige der oben vorgestellten AMLETO-Erweiterungen näher erläutert.

Der *control module generator* erzeugt zwischen den Verbindungen der einzelnen Komponenten sogenannte switches. Mit diesen kann während der Simulation ein Komponentenverhalten propagiert oder isoliert werden. Typischerweise beginnt man beim System-Design mit einem Top-Down-Entwurf. Mit zunehmender Verfeinerung der einzelnen Komponenten steigt auch deren Simulationszeit. Da man oft nur einen Teil des Systems näher untersucht und bereits in ihrem Verhalten bekannte Komponenten nicht fein aufgelöst simulieren muss, lässt sich mit dem Einsatz von switches eine Kombination verschiedener Sub-Architekturen erstellen. Die Schnittstelle zwischen den switches und der Außenwelt bildet der switch manager. Da die Erweiterungen als SystemC-Code implementiert werden, sind sie kompilierbar und somit kann das ausführbare Programm ohne Neuerstellung diese verschiedenen Sub-Architekturen je nach Bedarf ausführen.

Der *error list generator* extrahiert aus der fehlerfreien IIR-Darstellung die Fehlerliste. Hierzu gibt es für jede Basisanweisung eine Fehlerinjektionsregel, mit deren Hilfe eine entsprechende Fehlerinjektionsregel für jeden IIR-Knoten aufgerufen wird. Ergebnis ist dann eine zusätzliche, fehlerhafte IIR-Darstellung.

Der *error injector* ist für die fehlerhaften Anweisungen im SystemC-Code verantwortlich. Fehlermodell ist das Einzelhaftfehlermodell (engl. *single-stuck-at*). Dies führt zu einer Bitfehlerüberdeckung (*bit coverage error metric*) und wurde gewählt, weil Haftfehler zwischen den verschiedenen Abstraktionsebenen stark miteinander korrelieren. Da sich die Fehler auch in funktionalen Beschreibungen wiederfinden

müssen, sind hier einige Programmanweisungen gezielt zu sabotieren. Bei AMLETO sind dies Zuweisungen, bedingte Sprünge und Schleifen. Die fehlerbedingenden Anweisungen werden in den IIR-Code eingefügt. Die Abbildung 5.6 zeigt die Ansteuerung der Fehlerinjektion. Hier werden z. B. mit dem input0, input1 die Komponenten selektiert und mit Hilfe des Error\_port die gewünschte Fehlerart gewählt.

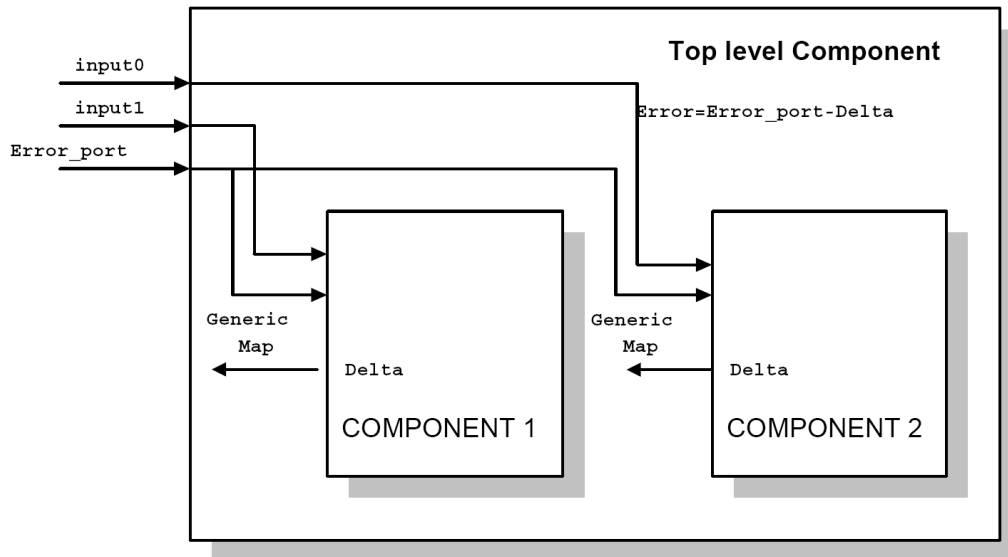


Abbildung 5.6: Architektur der Fehlerinjektion

Die *test pattern generation* verfolgt den Zweck eines inkrementellen Vorgehens. Hierzu werden zunächst Testsätze von vorherigen, abstrakteren Phasen genutzt und für nicht detektierte Fehler in der aktuellen Ebene neue Eingabevektoren erzeugt. In enger Wechselwirkung mit dem TPG befindet sich der *error simulator*. Er prüft die funktionale Äquivalenz in allen Designphasen, die bei einem Übergang zu einer niederen Ebene nötig ist.

Der *TPG generator* besteht aus drei wesentlichen Blöcken:

- **TPG:** Dieses Modul erzeugt die Eingabevektoren für das DUT. Bei der ersten Version von AMLETO war dies ein einfacher Zufallsgenerator. Später wurde die TPG erweitert, so dass schwer zu findende Fehler mittels genetischer Algorithmen ihre Stimuli erhalten.
- **Error detector:** Diese Komponente wird an das DUT angepasst und prüft die Signalausgabe.
- **Switch manager:** Dieser wählt die zu prüfende Sub-Architektur, indem er zwischen zwei Komponentenbeschreibungen (low level oder high level) innerhalb des DUT umschaltet. Die vorzunehmende Prozedur wird zur Laufzeit gelesen, wobei der Designer mit Hilfe einer Datei die Konfiguration festlegt.

Bei [119] wurde zunächst als Ergebnis gezeigt, dass eine Implementierung in SystemC durchaus sinnvoll und nützlich ist. Bei der etwas späteren Veröffentlichung in [38] wurde bei den vorgestellten Benchmark-Schaltungen deutlich, dass eine SystemC-Simulation gegenüber einer konventionellen VHDL-Simulation bezüglich der Simulationsgeschwindigkeit überlegen ist.

### 5.3.1.3 “High Level Fault Injection for Attack Simulation in Smart Cards”

Eine in [118] vorgestellte Arbeit beschäftigt sich speziell mit der Fehlerinjektion in Smart Cards. Smart Cards werden zunehmend in sicherheitsrelevanten Anwendungen verwendet. Dadurch kommt ihnen beim Schutz vor Attacken besondere Bedeutung zu. Oft wird diese Sicherheitsvalidierung erst recht spät im Entwurfsprozess vorgenommen und sie ist durch die entstandene Verfeinerung deshalb oft zeitaufwändig. In der vorgestellten Arbeit wird eine Fehlerinjektion auf einer höheren Ebene des Entwurfs vorgenommen. Dies erlaubt zum einen eine schon recht frühe Korrektur von Fehlern und zum anderen eine kurze Simulationszeit. Zudem gestattet die Repräsentation in SystemC einen flexiblen Wechsel zwischen verschiedenen Versionen. Ziel der Simulation ist es, eine Aussage zu treffen, ob und wie sich eine Fehlerattacke auswirkt. Angriffspunkte von Attacken können sich in dieser Arbeit entweder als Fehler auf Bussen, z. B. als Haftfehler oder als Bit-Flip-Fehler in Speichern, auswirken.

Für die Fehlerinjektion in Bussen werden zusätzliche fault injection modules (FIM) verwendet. Diese FIMs können zwischen den funktionalen Blöcken eingefügt werden. Die Abbildung 5.7 verdeutlicht dies. Die funktionalen Blöcke bleiben dabei ohne Modifikationen. Die Kontrolle der Fehlerinjektion erfolgt über fault injection control units (FICU). Die Fehlerinjektion in die FIMs selbst geschieht in Form einer Bitmaske per Parameterübergabe. Die FICU bestimmt Zeit, Ort und Anzahl der auftretenden Fehler. So ist auch eine Mehrfachfehlerinjektion möglich, wie sie bei Attacken beispielsweise auftritt.

Die Fehlerinjektion in Speicher geschieht mittels fault injection ports (FIP). Diese FIPs können interne Speicherbits manipulieren, ohne die Schreib- und Leseoperationen zu verändern. Diese Ports werden ebenfalls von der FICU kontrolliert. Die FIPs erlauben sowohl Haftfehler als auch transiente Fehler. Die FICU hat so viele Ports, wie die FIMs und die FIPs es erfordern. Bei großen Modellen führt dies zu einer erheblichen Mehrverdrahtung.



## 5.4 Fehlerinjektionstechniken in SystemC-Beschreibungen

Drei Techniken sind heute bei der simulierten Fehlerinjektion in HDL-Beschreibungen von Bedeutung. Dabei handelt es sich um die Saboteure (*saboteurs*), Mutanten (*mutants*) und den Simulatorkommandos (*simulator commands*).

Im Folgenden werden dazu eigene Implementierungen mit dem Fokus auf SystemC-Beschreibungen präsentiert. Je nach Technik ist die Fehlerinjektion für ganz spezielle Fehlerfälle geeignet. So eignet sich ein Saboteur gut für einen Haftfehler auf einer Signalverbindung und ein Mutant z. B. zur Injektion eines Fehlers an einem Gatterausgang. Durch Ähnlichkeitseigenschaften kann man aber einen Fehler aus der einen Technik in das Verhalten einer anderen Technik überführen. So verhält sich ein Haftfehler an einem Gatterausgang wie einer auf der angeschlossenen Leitung.

Simulierte Fehlerinjektionen hängen auch immer mit einem konkreten Fehlermodell zusammen. Dieses Fehlermodell nimmt in Abhängigkeit des zu modellierenden Defektes oder der vorliegenden Störung und der gewählten Entwurfsebene unterschiedliche Formen an. Eine ausführliche Aufstellung der im Zusammenhang zu verwendenden SystemC-Fehlermodelle findet man im Abschnitt 4.3.2.

In Bezug auf die Entwurfsebene wurden die Modelle mit einem Fokus für die Schalter-, Gatter- und Register-Transfer-Ebene entworfen. Sie sind aber auch in höhere Entwurfsebenen überführbar.

### 5.4.1 Die Saboteur-Technik

Saboteure sind zusätzliche Module, die sich im Signalweg zwischen zwei Komponenten wiederfinden. Ein Saboteur ist dabei selbst wieder ein Modul. Neben den Ein- und Ausgängen existiert mindestens ein Kontrolleingang, über den die Fehlerinjektion ausgelöst wird. In Abbildung 5.8 ist dies vereinfacht schematisch zu sehen.

Ein Saboteur verändert einen logischen Wert oder das zeitliche Verhalten einer Signalübertragung. Es ist auch eine Kombination von beidem möglich. Eine einfache Wertänderung entspräche z. B. einem permanenten Haftfehler und eine zeitliche Manipulation wäre z. B. eine einfache Verzögerung. Eine Kombination könnte eine transiente Änderung des Logikwertes sein.

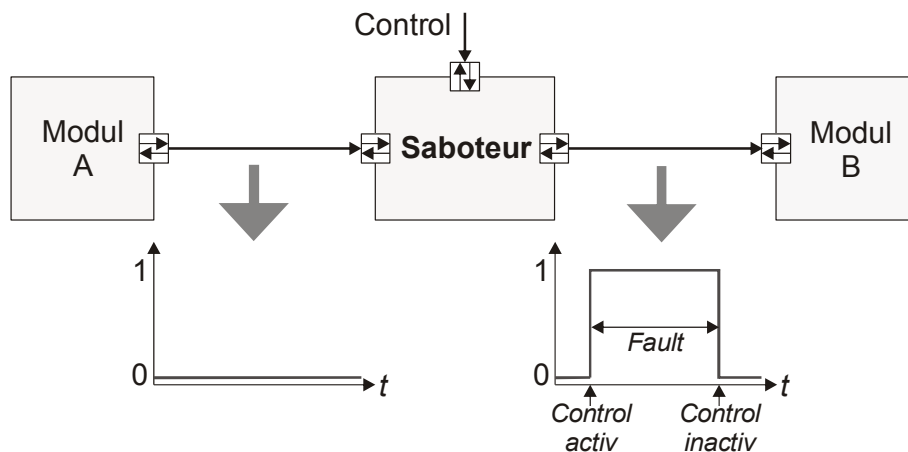


Abbildung 5.8: Prinzipielle Wirkungsweise eines Saboteurs

Von der prinzipiellen Struktur werden die Saboteure hier in drei Kategorien unterschieden. Diese sind:

- der einfache serielle Saboteur,
- der komplexe serielle Saboteur und
- der parallele Saboteur.

Ähnliche Einteilungen finden sich auch in [132,124]. Außerdem sind Kombinationen und hierarchische Organisationen dieser Modelle denkbar.

Der *einfache serielle Saboteur* wird in eine Signalleitung zwischen zwei Ports, die typischerweise zu jeweils einer Komponente gehören, geschaltet. Mit Hilfe dessen lassen sich alle Fehler injizieren, die auf einer Leitung denkbar sind. In den meisten Fällen genügt eine unidirektionale Verbindung, die Abbildung 5.9 stellt aber der Vollständigkeit halber eine bidirektionale Verbindungsstruktur dar.

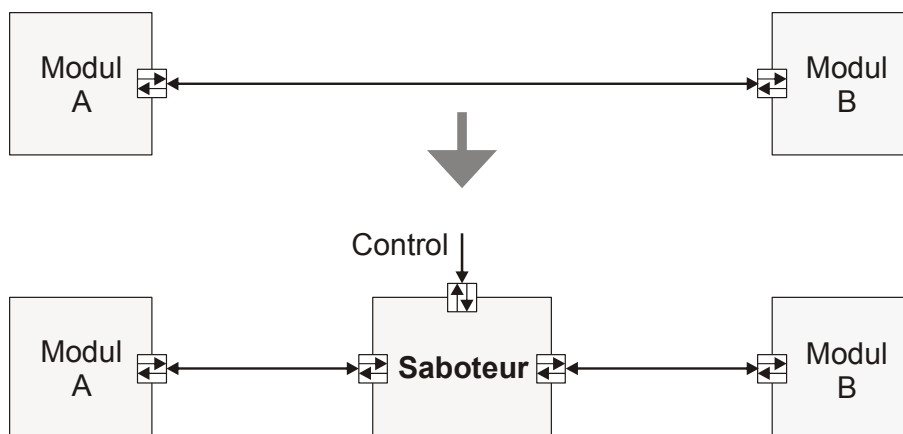


Abbildung 5.9: Einfacher serieller Saboteur



Das Listing 5.1 zeigt den Programmcode eines universellen seriellen Saboteurs. Zur besseren und kürzeren Übersicht arbeitet er unidirektional. Mit dem Saboteur lassen sich folgende Fehler injizieren:

- ein Haftfehler auf '0' bzw. stuck-at-0 (Zeile 20),
- ein Haftfehler auf '1' bzw. stuck-at-1 (Zeile 21),
- ein Bit-Flip-Fehler (Zeile 22),
- ein Unbestimmt-Wert 'X' (Zeile 23),
- eine offene Leitung 'Z' (Zeile 24),
- eine Verzögerung vom Wert  $t\_delay = 2$  (Zeile 26).

Bei einem Wert vom Fehlerinjektionseingang  $fi = 0$  oder  $fi > 6$  wird dort kein Fehler injiziert. Sind bei einer Fehlerinjektion nicht alle Injektionsmöglichkeiten nötig, so können die entsprechenden Zeilen im Saboteur-Modell entfallen. Neben einem kompakteren Modell erhält man, bedingt durch weniger Abfragen, eine schnellere Simulation.

Für einen bidirektionalen Saboteur sind die Dateneingänge und -ausgänge sowohl für Schreib- als auch Lesezugriff zu ändern. Die Entscheidung über die Datenrichtung wird häufig über einen zusätzlichen Kontrolleingang realisiert [124]. Alternativ kann der Fehlerinjektionseingang (→Listing 5.1, Zeile 2) diese Aufgabe auch übernehmen, da der verwendete Wertebereich mehr als ausreichend Platz bietet, um diese zusätzliche Information in das Modul zu führen.

In der Auswertung muss man nur noch einen zusätzliche Entscheidungsblock (→Listing 5.1, Zeile 16-26) für das umgekehrte Schreiben aufnehmen. Ein Beispiel für einen bidirektionalen Saboteur befindet sich im Anhang A.1.6.

---

```
1  . . .
1  SC_MODULE(saboteur_simple) {
    sc_in<sc_logic> in;
2  sc_in<int> fi;
    sc_out<sc_logic> out;
4  sc_time t_delay(2, SC_NS);

6  void pertub();

8  SC_CTOR(saboteur_simple) {
    SC_METHOD(pertub);
10     Sensitive << in << fi;
    }
12 };
    . . .
```

---

---

```

14 void saboteur_simple::pertub() {
    int v_fi=fi.read();    //liest das Fehlerinjektionkommando
16   switch (v_fi) {        //Fehlerinjektion
        case 0: out.write(in.read()); break;    //normal
18        case 1: out.write(sc_logic_0); break;  //stuck-at 0
        case 2: out.write(sc_logic_1); break;  //stuck-at 1
20        case 3: out.write(~(in.read())); break; //bit-flip
        case 4: out.write(sc_logic_X); break;  //Indetermination 'X'
22        case 5: out.write(sc_logic_Z); break;  //open line 'Z'

        case 6: next_trigger(t_delay);
24                out.write(in.read()); break;  //verzögert
        default: out.write(in.read()); break;  //normal
26   }
}

```

---

Listing 5.1: SystemC-Code für einen seriellen Saboteur

Der **komplexe serielle Saboteur** wird in Signalleitungen mit mehr als zwei Ports und u. U. auch zwischen mehreren Komponenten geschaltet.

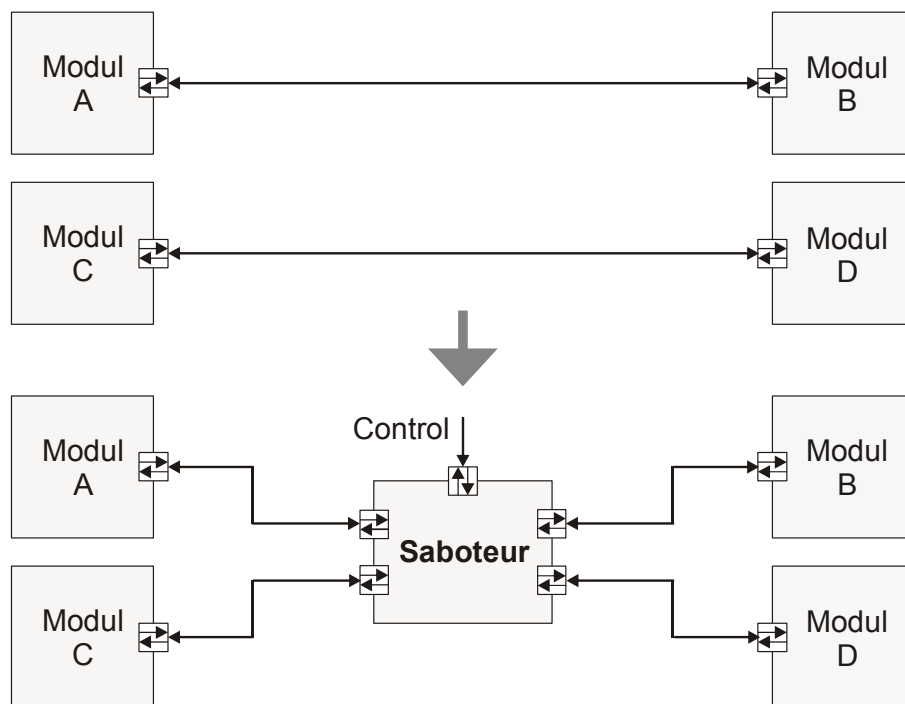


Abbildung 5.10: Komplexer serieller Saboteur

Mit diesem Saboteur werden solche Fehler injiziert, die zwischen mehreren Signalleitungen auftreten können. Aus physikalischer Sicht könnte das ein Defekt sein, welcher einen Brückenfehler (engl. *bridging fault*) hervor ruft (→4.3.2.1). Daneben kommen auch nicht ursächlich physikalische Fehler wie die intermittierenden vor. Ein Beispiel ist das Signalübersprechen. Solche Fehler sind zwar nicht permanenter Natur, aber reproduzierbar.

Ebenso kann man mit den komplexen seriellen Saboteuren auch Fehler auf Einzelleitungen injizieren, so wie beim einfachen seriellen Saboteur. Praktisch ist dies beim Umgang mit Busverbindungen. Ein solcher Saboteur wird auch als n-Bit-Saboteur bezeichnet. Ein n-Bit-Saboteur kann als ein einziger Anweisungsblock implementiert sein oder aber aus weiteren Saboteuren bestehen, die dann über mehrere Kontrolleingänge angesprochen werden.

Das Listing 5.2 zeigt für einen komplexen seriellen Saboteur den Ausschnitt der Fehlerinjektionsmethode.

---

```
2  . . .
   //x0,x1 = Eingänge; y0,y1 = Ausgänge
   void saboteur_br::pertub()
4  {
      if (fi.read()) {
6      if (x0.read()==x1.read()) {
          y0.write(x0.read());
8          y1.write(x1.read()); }
      else {
10         if ((x0.read()==sc_logic_X) || (x1.read()==sc_logic_X)) {
            y0.write(sc_logic_X);
12            y1.write(sc_logic_X); }
          else {
14             if ((x0.read()==sc_logic_Z) || (x1.read()==sc_logic_Z)) {
                 if (x0.read()==sc_logic_Z) {
16                     y0.write(x1.read());
                     y1.write(x1.read()); }
                 else {
18                     y1.write(x0.read());
                     y0.write(x0.read()); } }
20             else {
                 y0.write(sc_logic_X);
22                 y1.write(sc_logic_X); }
24         }
      }
26 }
   else {
28     y0.write(x0.read());
       y1.write(x1.read());
30 }
}
```

---

Listing 5.2: SystemC-Code für komplexen seriellen Saboteur mit Signaldominanz

Die Implementierung ist hier auf einer universellen Art mit bedingten Anweisungen durchgeführt. Die Funktion kann auch durch andere Wege definiert sein, z. B. mittels einer Tabelle. Ist die Fehlerinjektion *fi* aktiv, so werden die Ausgangssignale bei gekoppelten Leitungen wie bei *sc\_signal\_resolved* [42] geschrieben. Für den Fall kollidierender Signale erscheint meist ein unbestimmter Wert 'X' an den Ausgängen (Zeile 22,23). Bei einer nicht aktiven Injektion werden die Leitungen unabhängig voneinander behandelt (Zeile 28, 29). Dieses Modell findet sowohl für permanente

Fehler, wie sie durch physikalische Defekte entstehen, als auch für intermittierende Fehler Verwendung.

Ein anderes, häufig anzutreffendes Verhalten bei einem Überbrückungsfehler ist das Dominanzverhalten eines Signalwertes. Modelliert wird solch ein Verhalten als Wired-OR oder Wired-AND. Bei einer dominierenden, logischen '1' erfolgt eine OR-Verknüpfung der Signalleitungen. Die entsprechende Operation mit den Signalen  $x$  und  $y$  führt zu  $x'$ ,  $y'$  und ist wie folgt:

$$x' = y' = (x \vee y)$$

Das Auftreten einer einzigen '1' wirkt dann auf alle betreffenden Verbindungen. Bei einer dominanten logischen '0' wird eine AND-Verknüpfung benutzt.

$$x' = y' = (x \wedge y)$$

Das Listing 5.3 zeigt die Verknüpfung für einen signaldominierenden Saboteur.

---

```

2  case 7: out[0].write(in[0].read() & in[1].read()); // "0"-dominant
        out[1].write(in[0].read() & in[1].read());
4      break;
        case 8: out[0].write(in[0].read() | in[1].read()); // "1"-dominant
6      out[1].write(in[0].read() | in[1].read());
        break;
8  . . .

```

---

Listing 5.3: SystemC-Code für einen komplexen seriellen Saboteur

Die Verwendung einer universellen Implementierung nach Listing 5.2 lässt sich auch derart gut anpassen, dass eine Signalleitung eine größere Treiberstärke besitzt als die Nachbarleitung. Somit wird ein Dominanzverhalten im Ort statt im Wert erreicht. Weitere Informationen zu den realisierten Fehlermodellen findet man im Abschnitt 4.3. Eine Kombination mit Verzögerungsfehlern ist ebenso sinnvoll, da Überbrückungsfehler, bedingt durch höhere Lasten, zusätzliche Verzögerungen erzeugen.

Der *parallele Saboteur* wird an mindestens eine bestehende Signalleitung angefügt. Durch ihn lassen sich beispielsweise auf einem Bus Störungen injizieren, wie sie bei transienten Fehlern auftreten. Die Fehlerinjektion erfolgt dadurch, dass auf den gewünschten Leitungen das zusätzliche Modul Schreibvorgänge ausführt.

Das Listing 5.4 zeigt an einigen Beispielzeilen den zusätzlichen Einbau des Saboteurs (Zeile 9) und die Anbindung an einen bestehenden Bus (Zeile 10, 11).

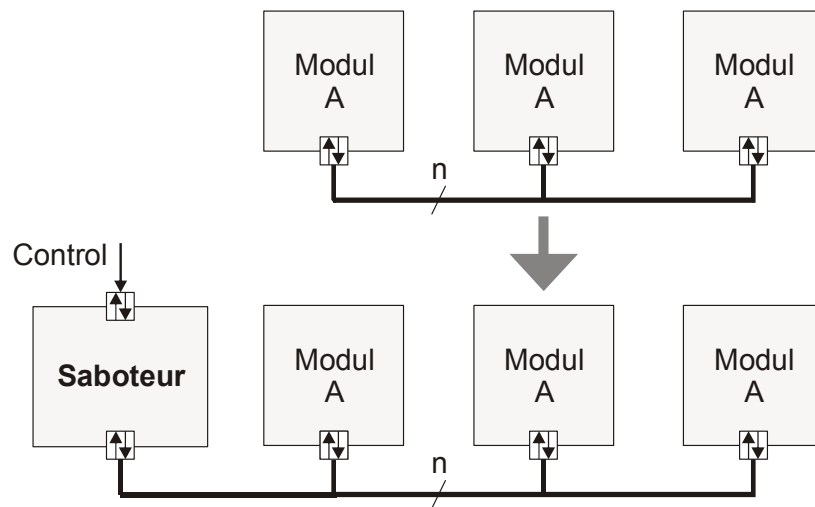


Abbildung 5.11: Paralleler Saboteur

---

```

2      . . .
      sc_signal<sc_lv<8> > BUS_A, BUS_B;
      sc_signal<int> SIG_FI;
4      . . .
      modul_shift M2 ( "M2" );
6      M2.x0(BUS_A);
      M2.y0(BUS_B);
8
      saboteur_br M3 ( "M3" );
10     M3.x0(BUS_A);
      M3.y0(BUS_B);
12     M3.fi(SIG_FI);
      . . .

```

---

Listing 5.4: Paralleler Saboteur - Instanziierung

Mit den vorgestellten Varianten entstand eine Bibliothek von Saboteuren. Mit den präsentierten Klassifizierungen und den gewählten Implementierungen ist diese Saboteur-Bibliothek  $S$  ein 4-Tupel  $S = (R, T, V, K)$  wobei:

- $R$  die Richtung -  $R \in \{\text{links, rechts, bidirektional}\}$ ,
- $T$  der Signal- /Datentyp -  $T \in \{\text{int, sc\_logic, ...}\}$ ,
- $V$  die Vektorisierung bzw. Parallelität -  $V \in \{\text{array, sc\_lv, ...}\}$ ,
- $K$  die Komplexität bzw. Verknüpfung von  $V$  -  $K \in \{1, ..., n\}$

ist.

### 5.4.1.1 Die Simulationsvorbereitung für Saboteure

Die Verwendung von Saboteuren erlaubt eine Reihe von Fehlerinjektionen und um eine aussagefähige Antwort zum Fehlerverhalten einer Schaltung zu bekommen, wird fast immer eine umfangreiche Anzahl an Saboteuren nötig sein. Bei kleinen Schaltungsmodellen ist die Simulationsvorbereitung noch manuell machbar, aber schon bei Schaltungen mit einigen hundert Gattern wächst der Arbeitsaufwand beträchtlich. Andererseits weist der Vorgang des Einfügens von Saboteuren immer wiederkehrende, ähnliche Verrichtungen auf. Dies war Anlass, die Simulationsvorbereitung für eine Simulation mit Saboteuren weitgehend zu automatisieren. Zur Schaltungsbearbeitung wurden dafür Programme in der Skriptsprache Perl [184] erstellt. Gemessen am Kompilervorgang in der C++-Umgebung war die Modellvorbereitung mit dem Perl-Skript äußerst gering und somit auch sehr effizient.

Die Abbildung 5.12 zeigt den Algorithmus der Simulationsvorbereitung mit Saboteuren in Form eines Pseudo-Codes.

---

```
FUNCTION AddSaboteur(DUT)
2  READ der Ein-/Ausgänge, Signale in temp. Variablen & DUT-File
   MAKE DUT-Kontrolleingänge zur Ansteuerung der Saboteure
4  IF Saboteure NOT Out-Saboteure THEN
   MAKE neue Saboteur-Ausgangssignale
6  ELSE
   MAKE neue Saboteur-Eingangssignale
8  IF Saboteure NOT Out-Saboteure THEN
   CONNECT Module mit Saboteur-Ausgangssignale
10 ELSE
   CONNECT Module mit Saboteur-Eingangssignale
12 MAKE Saboteur-Deklarationen
   IF Saboteure NOT Out-Saboteure THEN BEGIN
14   MAKE Saboteur-Instanzen
      CONNECT Saboteur-Eingänge mit DUT-Eingänge bzw. Signalen
16   CONNECT Saboteur-Ausgänge mit neuen Saboteur-Ausgangssignale
      CONNECT Saboteur-Kontrolleingänge mit neue DUT-Eingänge
18 END
   ELSE BEGIN
20   MAKE Saboteur-Instanzen
      CONNECT Saboteur-Eingänge mit neuen Saboteur-Eingangssignale
22   CONNECT Saboteur-Ausgänge mit DUT-Ausgänge
      CONNECT Saboteur-Kontrolleingänge mit neue DUT-Eingänge
24 END
   WRITE neues DUT-File
26 END FUNCTION
```

---

Abbildung 5.12: Pseudo-Code für DUT-Saboteur-Modifikation

Saboteure werden beim DUT anhand struktureller Gegebenheiten in zwei Kategorien unterteilt. Die sogenannten In-Saboteure befinden sich nach den Eingängen und zwischen zwei Modulen. Sie bekommen ihre Eingangsdatensignale aus der bestehenden Liste des ursprünglichen DUT. Die sogenannten Out-Saboteure befinden sich vor den

Ausgängen des DUT. Diese bekommen neue Signale an ihre Dateneingänge. Zusätzlich wird berücksichtigt, dass für Komponenten, die direkte Verbindungen zu primären Ein- oder Ausgängen haben, keine Signale existieren. Das DUT wird nach außen hin um die Ansteuerungsports der Saboteure erweitert.

Neben der Modellbearbeitung werden auch einige Dateien erzeugt, die als Schablone für eine Testbench dienen. Per Voreinstellung werden so schon eine fertige Testpattern-Generierung zum Anlegen der Schaltung mit Eingangsdaten und Anweisungen zum Aufzeichnen von Ausgaben generiert. Außerdem wird bei der Testbench-Erstellung Rücksicht auf die zu kontrollierenden Saboteure genommen. Diese werden sequenziell aus der Testbench heraus angesteuert. Es erfolgt somit eine Einzelfehlersimulation.

Die Abbildung 5.13 zeigt eine Übersicht über die erstellten Dateien nach der Simulationsvorbereitung.

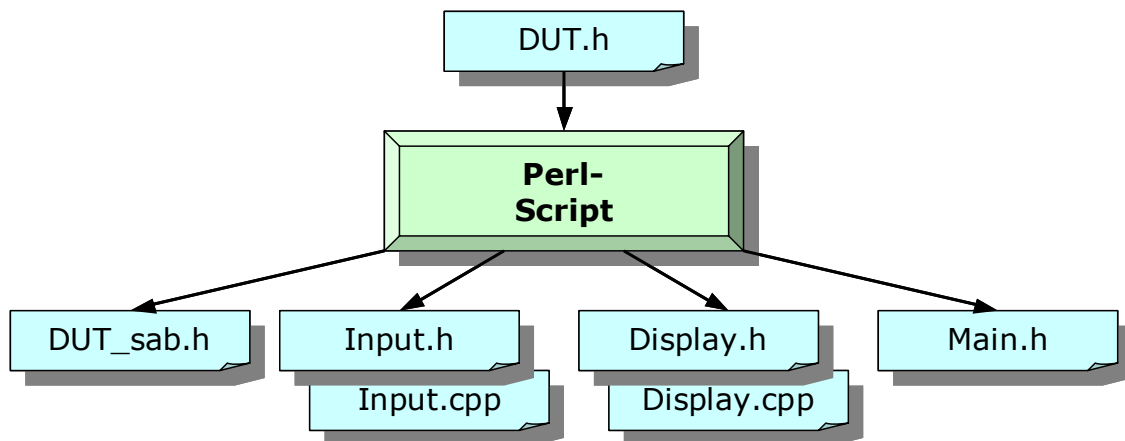


Abbildung 5.13: Testbench-Dateien nach der Simulationsvorbereitung

Nach der Bearbeitung steht das modifizierte Schaltungsmodell als *DUT\_sab* bereit. Die *Input*-Dateien versorgen die Schaltungseingänge mit Daten und kümmern sich auch um die Ansteuerung der Saboteure.

Die *Display*-Dateien beschäftigen sich mit der Ausgabe der Daten. In der *Main* werden all diese Module instanziiert und verknüpft. Ebenso stehen dort die Anweisungen, die für die Simulationssteuerung notwendig sind.

## 5.4.2 Die Mutant-Technik

Allgemein ausgedrückt bedeutet das „Mutieren“ innerhalb eines Programms das Modifizieren von Zeilen bzw. Anweisungen, die dann zu einem abweichenden Verhalten führen. Ursprünglich ist es eine Methodik, welche aus dem Software-Test bekannten Verfahren verwendet (→5.3). Im Anhang A.2 sind entsprechende Methoden aufgeführt. Da diese Verfahren vielmehr für verhaltensorientierte Modelle geeignet sind und deren Auflösung für genauere Untersuchungen in Hardware-Schaltungen nicht

reicht, werden diese nicht näher untersucht. Ein Beispiel, welches auch mit SystemC-Beschreibungen arbeitet und aus Verhaltensmodellen eine neue Bit-Überdeckung liefert, wurde in [119] vorgestellt (→5.3.1.2).

Im Allgemeinen versteht man unter einem Mutanten bei Nutzung einer Hardwarebeschreibungssprache, dass eine Komponente gegen eine andere ausgetauscht wird [124,132]. Für SystemC bietet sich dieses Verfahren besonders gut an, da es basierend auf C++ objektorientierte Strukturen hat. Da Komponenten als Objektinstanzen modelliert sind, kann man durch die Modifikation eines ursprünglichen Objektes die neuen Eigenschaften für alle betreffenden bzw. gewünschten Komponenten übernehmen.

Bei den Mutanten kann man nach [132] in vier Kategorien unterscheiden:

1. der einfache Austausch einer Komponente gegen eine mit anderer Funktionalität,
2. der Austausch gegen eine Komponente mit erweiterten Eigenschaften, die mehrere Funktionen gestattet,
3. das Hinzufügen eines Saboteurs in eine Komponentenbeschreibung,
4. das automatische Ändern von Anweisungen nach den Methoden des Software-Testens.

Der Punkt 4 wird aus den schon oben genannten Gründen (→5.3), wie eine zu ungenaue Fehlermodellierung, in dieser Arbeit nicht näher untersucht.

Mit der Bildung von Hierarchien, durch die Möglichkeit dass Module weitere Module (Submodule) enthalten können, ist der unter Punkt 3 aufgeführte Weg auch in SystemC durchführbar. Wenn das Fehlerverhalten auch innerhalb einer gekapselten Komponente wirkt, so ist die Durchführung der Fehlerinjektion mit der vorgestellten Saboteurmethode nahezu identisch. Deshalb wird hierzu auf die Saboteurmethode verwiesen (→5.4.1). Die beiden ersten Methoden werden im Folgenden vorgestellt.

Beim **einfachen funktionalen Austausch** wird die Fehlerinjektion dadurch erreicht, dass sich die Funktionalität ändert. So ergibt sich zum Beispiel aus einem AND-Gatter ein NAND-Gatter.

$$f(x, y) = z = (x \wedge y) \Rightarrow f'(x, y) = z' = \overline{(x \wedge y)}$$

In der Abbildung 5.14 ist dies dargestellt. Ursache der neuen Funktionalität kann z. B. ein Brückenfehler sein, wie er in der Abbildung 5.14 bei den beiden Ausgangstransistoren zu sehen ist. Diese Fehlerinjektion ließe sich im SystemC-Code durch eine Änderung der Komponenteninstanziierung vollführen. Solch eine Änderung kann also nur in der textuellen Beschreibung erfolgen und nach der Fehlerinjektion müsste das Modell neu kompiliert werden. Ein dynamisches Austauschen während der



Laufzeit ist in SystemC nicht möglich. Für die sequenzielle Injektion verschiedener Fehler in einem Simulationslauf, wie sie in der Praxis üblich ist, würde dies für jeden neuen Fehler ein neues Kompilieren und Linken bedeuten. Der Zeitbedarf dieser Vorgänge ist in C++ oft nicht unerheblich. Deshalb hat diese Methode in SystemC kaum eine praktische Bedeutung.

Alternativ zur eben beschriebenen Variante ließe sich die Austauschfunktionen innerhalb einer duplizierten Schaltung implementieren. Bei Fehlerinjektion würde man dann einfach die duplizierte Schaltung simulieren und so die zusätzliche Zeit für das Kompilieren und Linken sparen. Für ein Experiment wäre die Anzahl der Schaltungen  $S$  für eine Anzahl gegebener Fehlermodelle  $F_M$  und einer Anzahl von  $M$  zu untersuchenden Komponenten:

$$S = F_M \cdot M + 1 \quad (5.1)$$

Mit diesem Verfahren entsteht einerseits eine längere Simulationszeit durch das größere Modell und andererseits ein rapide wachsender Speicherbedarf durch das wiederholte Duplizieren der Schaltung. Für größere Schaltungen mit vielen Fehlerinjektionen ist dieser Weg problematisch.

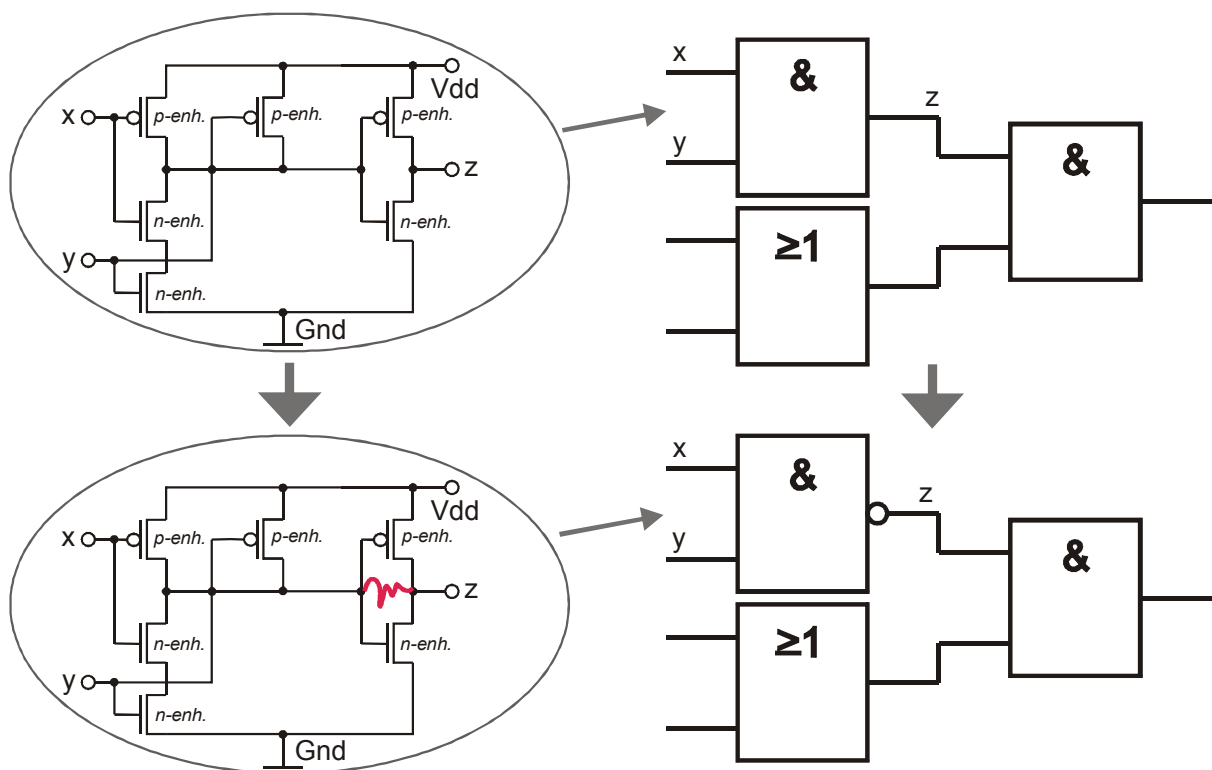


Abbildung 5.14: AND/NAND-Mutant durch Austauschen

Eine andere Art der Mutation ist der Austausch einer **Komponente** gegen eine **mit erweiterten Eigenschaften**. Diese Erweiterungen dienen zur Fehlermodellierung und Fehlerinjektion.

In der Abbildung 5.15 ist dies am Beispiel eines Speichermoduls dargestellt. Links befindet sich das Original-Modul und rechts ist das erweiterte Modul zu sehen.

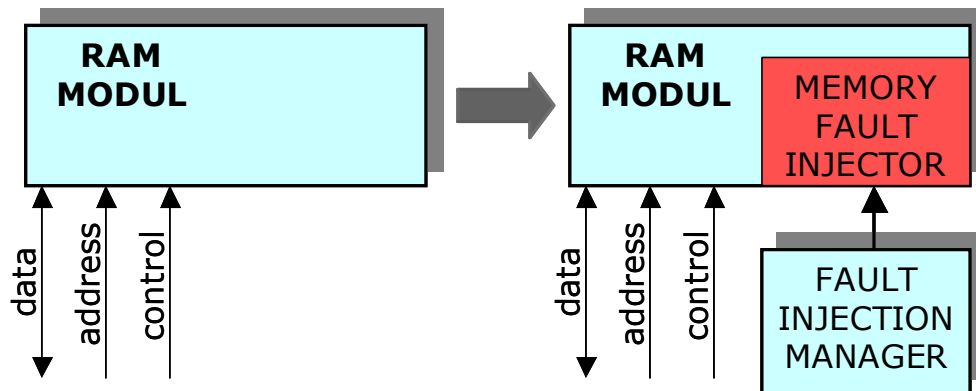


Abbildung 5.15: Erweiterter RAM-Mutant

Die Fehlerinjektion eines Mutanten wird typischerweise von außen gesteuert. In der Abbildung 5.15 ist dies durch den Fault Injection Manager verdeutlicht. Alternativ kann die Aktivierung und Deaktivierung auch innerhalb der Komponente stattfinden. Die Idee gab es schon einmal bei [146], wozu jedoch der VHDL-Sprachumfang erweitert werden musste. Dies kann man auch mit SystemC-Mitteln realisieren. Eine integrierte Takt- oder Simulationszeitabfrage kann nach einer gewünschten Zeit die Fehlersimulation auslösen und auch wieder aufheben. Vorteil ist hierbei der entfallende „Verdrahtungsaufwand“ im Programm, jedoch erkauft man sich die Reaktionsfähigkeit und die entstehenden Abfragen durch eine zunehmende Simulationszeit.

Die in dieser Arbeit verwendeten Mutanten nutzten die Idee, dass in einer Komponente durch zusätzliche Programmzeilen sowohl auf eine gewünschte Injektionsaktivierung reagiert werden kann, als auch die notwendigen Fehlermodelle (→4.3.2) implementiert sind. Die Umsetzung erfolgte manuell, d. h., es wurden Klassen von *fehlerhaften* Komponenten mit gegebenen Fehlermodellen erstellt. Durch die zu modellierenden Entwurfsebenen wie Schalter-, Logik-, RT-Ebene war die Realisierung der Komponenten mit ihren Schnittstellen gegeben und auch überschaubar. Die erstellte Mutanten-Bibliothek bot somit das Konzept der Wiederverwendung für weitere Projekte. Die Vorteile liegen gegenüber einer zufällig erzeugten Fehlerinjektion (*randomized fault injection*) in der genauen Einstellbarkeit bezüglich Ort, Anzahl und Zeit. Ein weiterer Gewinn dieser Mutanten-Variante ist die Modellierung nahezu beliebiger Fehler. Durch den reichhaltigen Sprachumfang von C++/SystemC ist dieser Ansatz universell nutzbar. Es sind sowohl die Manipulationen von Signalwerten als auch von Variableninhalten möglich. Ein weiterer Nutzen ist, dass der Programmcode

dieses erweiterten Mutanten in einer kompakten Form vorliegt, im Vergleich zum Mutanten, welcher durch einen zusätzlichen Saboteur entsteht.

Das Listing 5.5 zeigt ein Beispiel für einen Mutanten, der das Ausgangssignalverhalten manipuliert. Im Anhang A.1.1 ist zum Vergleich ein einfaches AND2-Modul abgebildet.

---

```

1  #include "systemc.h"
2
3  SC_MODULE (and2_mut) {
4      sc_out<sc_logic> z0;
5      sc_in<sc_logic> a0, a1;
6      sc_in<int> fi;
7
8      void and2calc();
9
10     SC_CTOR (and2_mut) {
11         SC_METHOD (and2calc);
12         sensitive <<a0<<a1<<fi;
13     }
14 };

```

---

```

15 . . .
16 void and2_mut::and2calc() {
17     sc_time t_delay(2, SC_NS);
18     int v-fi = fi.read();
19
20     switch (v-fi) {
21         case 0: z0.write (a0.read() & a1.read()); break;
22         case 1: z0.write(sc_logic_0); break;
23         case 2: z0.write(sc_logic_1); break;
24         case 3: next_trigger(t_delay);
25                 z0.write (A0.read() & A1.read()); break;
26         case 4: z0.write (~ (A0.read() & A1.read())); break;
27         default: z0.write (A0.read() & A1.read()); break;
28     }

```

---

Listing 5.5: Mutant in SystemC

In der oberen Hälfte des Listings ist die Header-Datei mit den Deklarationen und dem Konstruktor `SC_CTOR` abgebildet. In der unteren Hälfte ist die Implementierung zu sehen. In Abhängigkeit vom Fehlerinjektionseingang `fi` erfolgt ein Fehlerfall. Folgende Fehlermodelle finden sich wieder:

- Haftfehler am Ausgang `z0` auf logisch '0' (stuck-at-0) in Zeile 22 bei `fi = 1`,
- Haftfehler am Ausgang `z0` auf logisch '1' (stuck-at-1) in Zeile 23 bei `fi = 2`,
- Gatterverzögerung hier um 2 ns in Zeile 24 und 25 bei `fi = 3`,
- Austausch der Funktionalität (AND zu NAND) in Zeile 26 bei `fi = 4`.

In allen anderen Fällen (`fi = 0`, `fi > 4`) arbeitet der Mutant normal ohne Fehlerinjektion. Werden nun weitere Fehlermodelle und -orte, wie z. B. an den

Eingängen, und auch weitere Dateneingänge implementiert, kann der Aufwand stark anwachsen. In solchen Fällen lohnt sich eine Einschränkung in der Fehlermodellierung.

Die Umsetzung der Fehlerabfrage erfolgte im Listing 5.5 als *switch-case*-Abfrage. Das Realisieren der Fehlermodelle kann jedoch auf vielfältige Weise geschehen. Es wurden drei Varianten untersucht:

1. die Funktionalität wird in einer Tabelle abgelegt,
2. über bedingte Abfragen wird die gewünschte Ausgabefunktion selektiert,
3. mit arithmetischen oder logischen Operationen erfolgt die Fehlerinjektion.

Bei einer *tabellarischen Funktion* befindet sich der zu liefernde Ergebniswert in einem Array. Die Eingangswerte einer Komponente dienen der Indizierung des Arrays. Für logische Werte, wie *sc\_logic* in SystemC, bietet sich dieser Weg an, da der Wertebereich ('0', '1', 'Z', 'X') beschränkt ist. Bei großen Wertebereichen der Indizes setzen physikalische Grenzen ein. Ein Vorteil der Tabellenform besteht in der kurzen und konstanten Ausführungszeit. Es ist zum Beispiel gleich, ob eine Tabelle 4 oder 256 Einträge enthält.

Die *bedingte Abfrage* ist eine universelle und anschauliche Methode (→Listing 5.5, Zeile 8-17). Jedoch ist nur eine beschränkte Anzahl an Abfragen, sowohl aus Platz- als auch aus Gründen der Rechenzeit sinnvoll. Da die Abfragen sequenziell durchgeführt werden, können die Laufzeitnachteile messbar sein.

Die Durchführung mittels *arithmetischer* oder *logischer Operationen* ist eine wirksame und effiziente Variante. Jedoch lassen sich nur solche Fehler modellieren, für die es einen äquivalenten Befehl gibt. Für die arithmetischen Operatoren wären es beispielsweise Fehler in einer Verhaltensmodellierung, z. B. liefert eine Addition ein falsches Resultat. Für die logischen Operatoren wären es Fehler, die sich auf der Logikebene wieder finden. Das klassische Haftfehlermodell ist hierfür typisch. So wird ein stuck-at-0-Fehler zweckmäßig mit einer AND-Anweisung und ein stuck-at-1-Fehler mit einem OR-Befehl realisiert.

Im Gegensatz zu den Saboteuren können Mutanten auch *Fehler in Variablen* erzeugen. Typische Fehler, die bei einer Schaltung in Variablen auftreten, sind solche in Speichern. Hierbei ist es zweckmäßig, den Speicher als ein Array zu definieren. Der Index des Arrays steht dabei für die Adresse. Das Listing 5.6 zeigt einen Ausschnitt aus einem Beispiel. In diesem Fragment löst der Injektionskontrolleingang  $\hat{fi}$  bei einem Wert von  $\hat{fi} = 1$  eine Fehlerinjektion beim Schreiben in den Speicher aus. Mit Hilfe des Fehlermodell-Eingangs  $\hat{fm}$  und durch eine OR-Operation werden die entsprechenden Bits im Speicherdatum auf '1' gesetzt. Das vollständige Speichermodul befindet sich im Anhang A.1.7.

---

```
2      . . . else {                                     //schreiben
          if (addr.read() < MEM_SIZE)
4              if (fi == 1) {
                    ram[addr.read()] = data.read();
6                    ram[addr.read()] |= fm.read();
                    }
8              else
                    ram[addr.read()] = data;
10     }
    . . .
```

---

Listing 5.6: Mutant in SystemC für Variablen-Manipulation in einem RAM-Modul

### 5.4.2.1 Parametrisierte Module

Eine leistungsfähige Eigenschaft in SystemC ist die Möglichkeit zur Unterstützung parametrisierter Entwürfe. Viel Aufwand wird häufig in die Entwicklung von Modulen und Kanälen (channels) gesteckt. Mit parametrisierten Modulen kann dieser Aufwand verringert werden. Diese Fähigkeit kennt man ebenso von anderen Sprachen, auch Hardwarebeschreibungssprachen, unter dem Begriff der generischen Programmierung. Es wird einmalig ein Modul erstellt und später einfach per Parameter an den erforderlichen Einsatzfall angepasst. Zum Beispiel ist es in SystemC möglich, eine Stack-Klasse zu kreieren, die gerade den Datentyp verarbeitet, den sie als Parameter bekommt. Es beherrschen auch andere HDLs - wie VHDL oder Verilog - Teile der generische Programmierung, aber dieser Anwendungsfall mit einem variierenden Datentyp als Parameter wird z. B. dort nicht unterstützt [35].

Die Möglichkeiten der Parametrisierung können verschiedener Art sein. Da in dieser Arbeit die Fehlerinjektionen in Form von Saboteuren und Mutanten zumeist mit Modulen und weniger mit Kanälen entstehen, wird im folgenden Text nur auf Module Bezug genommen. Die vorgestellten Prinzipien haben aber ebenso für Kanäle Gültigkeit. Im Folgenden ist eine sinnvolle Aufstellung aufgeführt:

- **Parametrisierte Design-Struktur:** Je nach Bedarf werden innerhalb des Moduls Funktionen aktiviert oder Funktionen erweitert. Bei einem Logikgatter kann das die Anzahl der Eingänge betreffen.
- **Parametrisierte Werte:** Innerhalb des Moduls werden Wertebereiche angepasst. So kann zum Beispiel der Adressraum eines Speichers als Parameter festgelegt werden.
- **Parametrisierte Typen bzw. Typ-Attribute:** Der zu verwendende Datentyp kann je nach Bedarf festgelegt werden. Dies kann z. B. als Typ der zu verwendende Logiktyp wie zwei- oder vierwertig sein. Andererseits lässt sich als Attribut auch die Genauigkeit einer Fixkommazahl bestimmen.

Nach [35] kann in SystemC die Parameterübergabe in drei Zeitstufen aufgelöst werden:

- **Zur Kompilierzeit:** Die Parameter werden vom C++-Compiler aufgelöst. Um die Parameter aufzulösen, nutzt man Templates.
- **Zur Elaborationsphase:** Die Parameter werden während des Aufbaus der SystemC-Hierarchie aufgelöst. Diese Ausarbeitungszeit liegt zwischen `sc_main()` und dem Simulationsstart `sc_start()` bzw. `sc_cycle`. Hierzu nutzt man C++-Konstruktor-Argumente.
- **Zur Simulationszeit:** Die Parameter werden während der Simulationszeit aufgelöst. Solch ein Parameter kann durch eine Member-Variable innerhalb eines Moduls realisiert sein. Eine öffentliche Methode erlaubt dann z. B. das dynamische Setzen des Parameters während der Simulation.

Die mögliche Variante der Parameterauflösung zur Simulationszeit wurde nicht näher betrachtet, da die typischen Parameter für den Fall der verwendeten Mutanten und Saboteure mit ihr unzureichend umzusetzen sind.

Die Parametervarianten mit Templates und Konstruktor-Argumenten wurden zum Zwecke der Fehlerinjektion implementiert und näher untersucht.

### 5.4.2.1.1 Parametrisiertes Template-Modul

Wie bereits oben erwähnt, erfolgt die Parameterauflösung während der Kompilierzeit mit Hilfe von Templates (dt. *Schablonen*). Templates sind Vorlagen für Algorithmen. Nützliche Parameter sind im Falle eines Mutanten der Typ eines Signals, die Anzahl von Ein- und Ausgängen oder die Größe einer inneren Struktur, wie z. B. die Größe eines RAM-Moduls. Die Vorteile des Template-Konzeptes sind die hohe Wiederverwendbarkeit, Anpassungsfähigkeit und die einfachere Komponentenverwaltung. Das Listing 5.7 zeigt ein Beispiel für einen Template-Mutanten. Dieses Modul ist eine AND-Funktion, bei der die Anzahl der Eingänge beliebig sein kann.

Mit dem *size*-Parameter in der Template-Anweisung (Zeile 2) wird die Anzahl der Eingänge übergeben. Ohne eine Festlegung beträgt sie voreingestellt zwei. Dieser Parameter wird bei der Deklaration der Eingänge benutzt (Zeile 4). In Abhängigkeit eines Fehlerkontrollwertes *v\_fi* wird die normale Funktion (Zeilen 17-20) oder eine Fehlerinjektion am Ausgang vorgenommen (Zeile 23). Die typischen Erweiterungen für die Template-Realisierung sind im Listing 5.7 hervorgehoben. Das vollständige AND-Modul befindet sich im Anhang A.1.8.

---

```
2  template <int size = 2>
   SC_MODULE(and) {
4      sc_in<sc_logic> a[size];

   . . .

6      SC_HAS_PROCESS(and);
      and(sc_module_name name_) : sc_module(name_)
8      {
          SC_METHOD(doit);
10         for (int i=0; i < size; i++) sensitive << a[i];
          . . .
12
      template <int size>
14 void and<size>::doit() {

   . . .

16     switch (v-fi) {
          case 0: while (i<size) {
18                 zt = zt & a[i].read();
                    ++i;
20             }
                z0.write(zt);
22             break;
          case 1: z0.write(sc_logic_0); break;      //sa0
24     . . .
}
```

---

Listing 5.7: Template-Mutant mit AND-Funktion

Im Listing 5.8 ist die entsprechende Instanziierung des Template-Moduls wiedergegeben. In der Zeile 1 entsteht das Objekt M3 und nachfolgend wird es mit den passenden Signalen verbunden.

---

```
and<3> M3("M3");
2  M2.a[0] (SIG_S0);
   M2.a[1] (SIG_S1);
4  M2.a[2] (SIG_S2);
   . . .
```

---

Listing 5.8: Instanziierung des Template-Mutanten

Positiv bei der Template-Anwendung ist, dass die Auflösung schon während des Kompilierens stattfindet. Das kostet deshalb keine Simulationszeit. In Abschnitt 5.4.4 sind einige Simulationsergebnisse zum Vergleich gegenübergestellt. Die vorgestellte Template-Technik wurde ebenso erfolgreich für Saboteure implementiert.

#### 5.4.2.1.2 Parametrisiertes Konstruktor-Modul

Die Parameterrauflösung mit Hilfe von Konstruktor-Argumenten erfolgt, wie oben erwähnt, während der Ausarbeitungsphase (Elaboration). Bei diesem Verfahren können nach der Kompilierung noch Änderungen an der Designstruktur stattfinden. So ist es vor der Simulation beim Programmaufruf per Argumentübergabe möglich, ein Modell wie gewünscht festzulegen. Andererseits können Konstruktor-Argumente keine C++-

Typen spezifizieren, wie es bei den Template-Parametern möglich ist. Die Variante mit den parametrisierten Konstruktoren ist somit etwas weniger praktisch als die Template-Variante. Im Listing 5.9 ist ein Beispiel für den AND-Mutanten mit Konstruktorparametern aufgeführt.

---

```
2  . . .
   SC_MODULE(and) {
4     sc_in<sc_logic> *a;
     sc_in<int> fi;
     int size;
6     . . .
     SC_HAS_PROCESS(and);
8     and(sc_module_name name_, int size_=2) :
       sc_module(name_) {
10        size = size_;
        in = new sc_in<sc_logic>[size];
12        SC_METHOD(doit);
        for (int i = 0; i < size; i++) sensitive<<a[i];
14        sensitive << fi;
       . . .
16  . . .
   void and::doit() {
18     int i, v-fi = fi.read();
     sc_logic zt = a[0].read();
20
     switch (v-fi) {
22         case 0: while (i < size) { //and3
                     zt = zt & a[i].read();
24                     ++i;
                     }
26         z0.write(zt);
                     break;
30         case 1: z0.write(sc_logic_0); break; //sa0
       . . .
   }
```

---

Listing 5.9: Konstruktor-Mutant mit AND-Funktion

Beim Beispiel im Listing 5.9 wird der Parameter *size* übergeben, der bezeichnet, wie viele Operatoren für die AND-Funktion zu verwenden sind (Zeile 22). Außerdem bestimmt der *size*-Parameter auch die Anzahl der Ports. Damit sich die Port-Anzahl auch nach der Kompilation ändern kann, werden die Ports dynamisch mit Hilfe von Zeigern erzeugt. Wie schon beim Template-Modul, muss beim parametrisierten Konstruktor der Standard-Konstruktor *SC\_CTOR* gegen *SC\_HAS\_PROCESS*, etc. getauscht werden. Die Besonderheiten sind in den Listings 5.9 und 5.10 hervorgehoben. Das vollständige Beispiel ist im Anhang A.1.9 zu finden. Das Listing 5.10 zeigt die Instanziierung des Konstruktor-Mutanten und dessen Verdrahtung.



---

```
. . .
2  and M2 ("M2", 3);
    M2.a[0] (SIG_S0);
4   M2.a[1] (SIG_S1);
    M2.a[2] (SIG_S2);
6   M2.fi (SIG_FT);
. . .
```

---

Listing 5.10: Instanziierung des Konstruktor-Mutanten

#### 5.4.2.1.3 Ergebnisse der Parametrisierten Module

Generell ist es empfehlenswert, die Parameterrauflösung so früh wie möglich vorzunehmen. Gründe hierfür sind:

- Entwurfsfehler können früher im Designprozess erkannt werden,
- die frühe Fehlererkennung kann ausgereifter sein, so können zum Beispiel Typinkompatibilitäten schon vom C++-Kompiler entdeckt werden,
- dem Kompiler ist u. U. eine bessere Codeoptimierung möglich.

Dennoch kann die Parameterrauflösung mit Konstruktor-Argumenten sinnvoll sein. Beim Umgang mit Intellectual Property (IP) und der Nutzung von C++-Templates müssen Programmteile für den Kompiler als offener Quellcode vorliegen. Dies erschwert den Schutz des Know-hows für den Lieferanten. Der Gebrauch von Konstruktor-Argumenten ist hierzu eine Alternative.

Der C++-Sprachstandard erlaubt nur einfache Typen für die Argumentenübergabe [35]. Bei dieser expliziten Spezialisierung [177] ist keine String-Übergabe oder keine Verwendung von Arrays möglich.

In der Tabelle 5.1 sind einige Simulationsergebnisse für Mutanten aufgeführt. Hierbei wurden verschiedene Modelle mit diversen Funktionen untersucht. Wiedergegeben ist in der Tabelle die Beschleunigung (*Speed-up*) in Bezug zur Referenz. Als Referenzen gelten die nicht-parametrisierten Implementierungen. Da die parametrisierten Modelle, bedingt durch die veränderliche Port-Anzahl, eine andere Implementierung der Funktion erfordern, können sich Abweichungen bei der Ausführungszeit ergeben. Ein Extremfall ergibt sich, wenn fast nur gerechnet wird. Dabei ist der Rechenanteil der veränderten Funktionen an der Gesamtausführung besonders hoch. In der Tabelle entspricht das den Werten ohne Ausgabe. Für die Beispiele mit den parametrisierten Modulen ergeben sich im Wesentlichen keine Nachteile. Die Lösung mit den Konstruktor-Argumenten ist mitunter sogar schneller als die übliche, nicht-parametrisierte Methode. Außerdem ist noch die Simulation mit Ausgabe wiedergegeben. Dieser Fall liegt näher an der Realität. Bei diesen Simulationen sind die Unterschiede zwischen den verschiedenen Ausführungen verschwindend gering.

Design	Speed-up					
	normales Modul		Template-Modul		Konstruktor-Modul	
	mit Ausgabe	ohne Ausgabe	mit Ausgabe	ohne Ausgabe	mit Ausgabe	ohne Ausgabe
decod	1,0	1,0	0,99	1,01	1,01	1,01
fsm6	1,0	1,0	1,001	1,057	1,036	1,117
c17	1,0	1,0	1,00	0,973	1,01	1,0

Tabelle 5.1: Simulationsergebnisse für (nicht-)parametrisierte Beispiele

### 5.4.2.2 Die Simulationsvorbereitung für Mutanten

Im Vergleich zum Einfügen von Saboteuren in die Schaltungsbeschreibung (→5.4.1) ist die Mutant-Methode weniger aufwändig. Bei großen Schaltungen wächst der Aufwand der Simulationsvorbereitung dennoch, so dass sie rein manuell nicht zu bewältigen ist. Dank ähnlich wiederkehrender Aufgaben ließ sich die Ausarbeitung automatisieren.

Die Abbildung 5.16 zeigt vereinfacht den Algorithmus der Schaltungsmodifikation. Wesentliche Änderungen sind der Austausch der ursprünglichen Komponenten gegen die Mutanten, das Hinzufügen zusätzlicher Eingänge in die Schaltungsbeschreibung zur Ansteuerung der Mutanten und die erforderliche zusätzliche Verdrahtung.

---

	<b>FUNCTION</b> MakeMutant(DUT)
2	<b>READ</b> Module in temporäre Variable & DUT-File
	<b>SWAP</b> originale Module gegen Mutanten in der Deklaration
4	<b>MAKE</b> DUT-Kontrolleingänge zur Ansteuerung der Mutanten
	<b>SWAP</b> originale Module gegen Mutanten in der Instanziierung
6	<b>MAKE</b> neue Mutant-Kontrolleingänge bei der Instanziierung
	<b>CONNECT</b> Mutant-Kontrolleingänge mit neue DUT-Kontrolleingänge
8	<b>WRITE</b> neues DUT-File
	<b>END FUNCTION</b>

---

Abbildung 5.16: Pseudo-Code für DUT-Mutant-Modifikation

Nach der Modifikation stehen zur Simulation die Testbench-Dateien bereit, wie schon in der Abbildung 5.13 dargestellt und unter 5.4.1.1. beschrieben.

### 5.4.3 Simulationskommandos

Einige Logiksimulatoren und auch HDL-Simulatoren besitzen zusätzliche Kommandos, die eine Manipulation von Signalen bzw. Variablen während des Simulationslaufes gestatten (→5.3). Meistens werden diese Kommandos als *simulator commands* (dt. *Simulatorkommandos*) oder *built-in commands* bezeichnet. Die Leistungsfähigkeit dieser Simulationsprogramme kann je nach Produkt stark differieren. SystemC bietet von sich aus kaum bzw. keine Unterstützung dieser Technik. Im Folgenden werden

einige Erweiterungen von SystemC gezeigt, die der Technik der *simulator commands* ähnlich sind. Mit den *simulator commands* existiert die Gemeinsamkeit, dass sie ähnlich einfach in ihrer Bedienung ist. Außerdem bedarf es selten einer Modifikation der Schaltung (DUT). Jedoch unterscheidet sich die Art der Implementierung der hier vorgestellten Techniken, weshalb in dieser Arbeit der Begriff der *Simulationskommandos* (engl. *simulation commands*) eingeführt wird.

Signale oder Variablen innerhalb eines Moduls lassen sich dann von außen beeinflussen, wenn sie in das Modul bzw. in die Datei mit den manipulierenden Anweisungen eingebettet sind. Das DUT ist somit ein Bestandteil der Fehlerinjektionskomponente bzw. das DUT ist ein Kindmodul. Zwei mögliche Implementierungen sind in der Abbildung 5.17 dargestellt. In der linken Hälfte (a) ist das DUT in einem sogenannten Injektionsmodul (Injection Modul) eingebettet. In Abhängigkeit von Eingangsmustern oder vom Simulationszeitpunkt führt es manipulierende Anweisungen aus. Eine Anweisung sieht zum Beispiel wie folgt aus:

```
ModulA.SubModul1.Var1 = 1;
```

In dieser Anweisung werden zunächst die Komponente, dann die Variable und anschließend der zu setzende Wert spezifiziert. In der rechten Hälfte wird das DUT direkt aus der Main-Datei (*main.cpp*) angesprochen. Neben der Instanziierung der Module finden sich in dieser Datei die manipulierenden Anweisungen wieder. Um eine Interaktion während der Simulation zu gewährleisten, wird der normale SystemC-Befehl *sc\_start()* zur Aktivierung der Simulation durch *sc\_initialize()* und einer Mehrfachverwendung von *sc\_cycle()* ersetzt. Im Listing 5.11 wird dies deutlich.

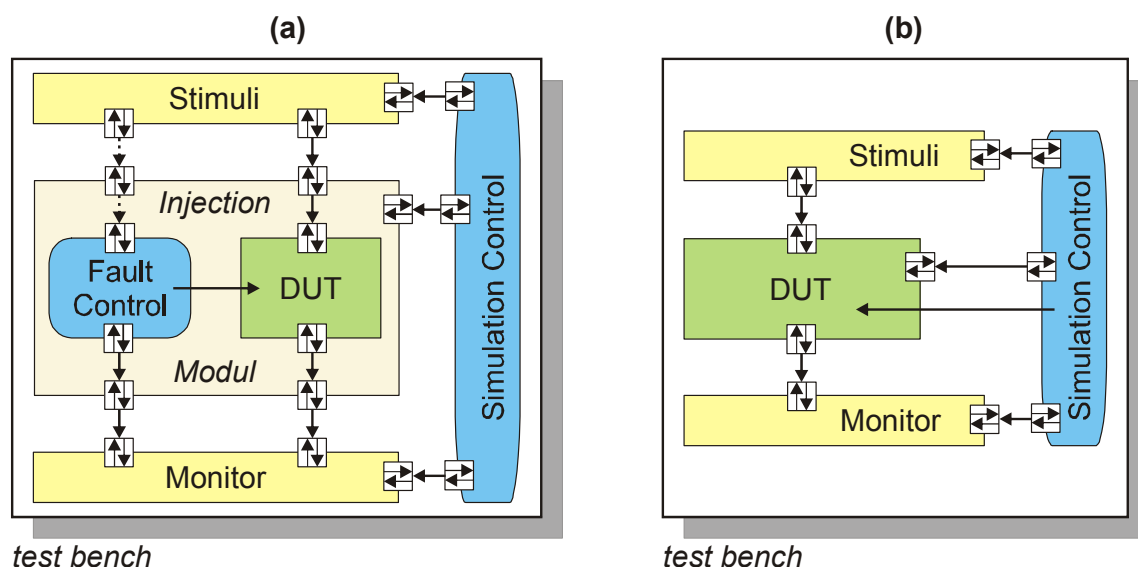


Abbildung 5.17: Allgemeine Injektion mit Simulationskommandos

Leistungsfähige HDL-Simulatoren bieten die Möglichkeit, Sequenzen von Simulator-Kommandos in Skripte zusammenzufassen. Während der Simulation wird dann das komplette Skript abgearbeitet. In einem SystemC-Projekt wäre ein solcher Weg ebenso denkbar. Hierzu müsste ein zusätzliches Objekt diese Skript-Anweisungen interpretieren und während der Simulation äquivalente C++/SystemC-Befehle dazu ausführen. Da jedoch ein SystemC-Projekt auch in einem lesbaren Quelltext vorliegt und vor einer Simulation kompiliert werden muss, scheint es angemessen, auf solch ein interpretierendes Zusatzmodul zu verzichten. Der direkte Weg führt zu weniger Simulationsaufwand.

### 5.4.3.1 Simulationskommandos mit Signalen

Die Fehlerinjektion in Signale beeinflusst zwangsläufig die Kommunikation zwischen zwei Modulen bzw. Prozessen. Ist ein Modul in einem anderen Modul gekapselt, so handelt es sich um ein Submodul. Die Signale eines Submoduls können aus einem übergeordneten Modul manipuliert werden, ohne dass es hierzu zusätzlicher Ports und Verbindungen bedarf. Dies ist weniger ein SystemC-Ansatz, sondern diese Eigenschaft entsteht durch den Gebrauch von C++. Auf diese Weise lässt sich mit einem einzigen Befehl ein Signal beeinflussen, die Syntax ist wie folgt:

```
Modulname.Signalname.write(Wert);
```

Üblicherweise wird dann das Signal diesen neuen Wert übernehmen. Kommt jedoch der aufgelöste Signaltyp *sc\_signal\_resolved* zum Einsatz, so entspricht der Signalzustand dem der Auflösungstabelle [42]. Trifft z. B. ein logisches '1' auf eine logische '0', so dominiert nicht die '1' sondern es entsteht ein „unbestimmtes“ 'X'.

Eine weitere Eigenheit ist bei dieser Art der Injektion und dem Nichtgebrauch von *sc\_signal\_resolved* zu berücksichtigen. Die Injektion dauert nur bis zum nächsten Schreibvorgang an, ganz gleich, aus welcher Quelle dieser Vorgang stammt. Dadurch ist eine definierte Injektionsdauer schwer zu realisieren. Die Injektion kann im nächsten Simulationszyklus schon wieder unwirksam sein oder sie dauert bis zum Simulationsende. Mit einigen Abstrichen an Realitätsnähe sind so Injektionen transienter Fehler auf einfache Weise möglich. Zur Injektion permanenter oder vorübergehender Fehler mit definierter Zeitlänge ist dieser Weg ungeeignet.

Bei der Signalmanipulation mit Simulator-Kommandos geht man nach [132] so vor, dass nach einer gewissen Simulationszeit die Simulation gestoppt wird, dann das Signal von seinem Treiber unterbrochen und auf einen neuen Wert gezwungen wird. Anschließend erfolgt für eine bestimmte Zeit mit dieser festen Signalstärke eine Simulation und wenn die Fehlerinjektion aufgehoben werden soll, wird das Signal wieder mit seinem ursprünglichen Treiber verbunden. Diese Vorgehensweise ist in SystemC problematisch. Es ist zwar in der neuen SystemC-Release [42] realisierbar,

Ports offen zu lassen, aber über die Möglichkeit eines dynamischen „Umverdrahtens“ während der Simulation ist nichts bekannt [35].

Um die Fehlerinjektion in SystemC mittels Kommandos zu erhalten, kam es zu einigen Änderungen an der SystemC-Bibliothek. Da man für die Simulation von Fehlern in digitaler Hardware meistens auch Logiktypen einsetzt, wurden diese Klassen überschrieben bzw. erweitert. Die Erweiterungen sind in der folgenden Übersicht zusammengefasst:

1. Der Datentyp *sc\_logic* wurden um zusätzliche Datenwerte ('B', 'A', 'R') erweitert.
2. Um die neuen Werte auch sinnvoll auszuführen, wurden die Operatoren und einige Methoden in der Klasse *sc\_logic* modifiziert.
3. Für eine brauchbare Typumwandlung der neuen Werte musste die Konvertierungstabelle überschrieben werden.
4. Den neuen Werten wurden in der Klasse *sc\_signal\_resolved* mit Hilfe einer Auflösungstabelle neue Signalstärken zugewiesen.
5. Weitere Methoden wurden in anderen Klassen angepasst. Dies betraf z. B. die Methode *sc\_trace()*, welche die Aufzeichnung von Signalen während der Simulation ermöglicht.

Für die Injektion von Haftfehlern wurde in dieser Arbeit zu den bestehenden Logikwerten '0', '1', 'Z' und 'X' die drei neuen Datenwerte '**B**', '**A**', '**R**' eingeführt. Auf dem Gebiet der ATPG hat sich für Berechnungen mit dem Haftfehlermodell die 1966 von Roth eingeführte 5-wertige Algebra etabliert [153]. Für eine einfache eindeutige Zuweisung in den SystemC-Klassen war diese D-Notation jedoch impraktikabel, weshalb eine neue Notation eingeführt wurde. Die Menge des erweiterten Logiktyps lautet:

$$sc\_logic \in \{0, 1, Z, X, B, A, R\}$$

bzw. nach SystemC-Syntax:

$$sc\_logic \in \{sc\_logic\_0, sc\_logic\_1, sc\_logic\_Z, sc\_logic\_X, sc\_logic\_B, sc\_logic\_A, sc\_logic\_R\}$$

Allein mit der Einführung zusätzlicher Werte in die Klasse *sc\_logic* ist noch keine Injektion permanenter Fehler möglich. Erst die Kombination der neuen Logikwerte mit neuen *Signalstärken* führt zu einem deterministischen Fehlerverhalten. Um das zu erreichen, wurde die Auflösungstabelle (*resolved table*) in der Klasse *sc\_signal\_resolved* erweitert. Die Tabelle 5.2 zeigt die erweiterte Auflösung.

Die Injektion des Wertes 'A' (Above) steht für eine „starke“ '1' und führt zu einem stuck-at-1-Fehler. Das Schreiben eines logischen 'B' (Below) meint eine „starke“ '0' und erzeugt einen stuck-at-0-Fehler. Das Anlegen von 'R' (Release) gibt das Signal wieder frei und hebt somit die Fehlerinjektion auf.

In der Tabelle 5.2 stehen die erste Zeile und die erste Spalte für die aufeinander treffenden Signalwerte. Das Resultat beider ergibt sich aus dem Tabelleninhalt. Bei näherer Betrachtung der Zeilen und Spalten von *A* und *B* lässt sich erkennen, dass diese ein dominantes Verhalten aufweisen. Nur durch ein *R* werden diese Signale wieder aufgehoben. Andererseits lässt sich erkennen, dass die normalen logischen Signale (0, 1, Z, X) eine höhere Priorität besitzen als das Release-Signale (*R*). Durch diesen Trick kann die Dominanz der Fehlerinjektion während der Simulation wieder aufgehoben werden. Dies erlaubt auch die Injektion mit einer gewissen Dauer, wie sie bei transienten Fehlern üblich ist.

	0	1	Z	X	B	A	R
0	0	X	0	X	B	A	0
1	X	1	1	X	B	A	1
Z	0	1	Z	X	B	A	Z
X	X	X	X	X	B	A	X
B	B	B	B	B	B	X	R
A	A	A	A	A	X	A	R
R	0	1	Z	X	R	R	R

Tabelle 5.2: Neue Auflösungstabelle in der Klasse *sc\_signal\_resolved*

Durch diese rotierende Priorität der Signalwerte könnte theoretisch unter ungünstigen Umständen ein unendlicher Simulations-Zyklus (→3.1.5.1) entstehen und würde sich durch ein „Hängen“ der Simulation äußern. Zwei Maßnahmen verhindern dies. Zum einen wird eine Fehlerinjektion an einem Fehlerort nur aus einem Prozess heraus unternommen (→Listing 5.11). Zum anderen ist ein Ausgangssignal nach einer Funktion (→Listing 5.11) nie ein Fehlerinjektionssignal vom Wert A oder B, so dass sich durch Signalfortschaltung auch kein unendlicher Simulationszyklus entstehen kann.

Um die neuen Werte sinnvoll verwenden zu können, mussten die *Operatoren* *&*, *|*, *^*, *~* erweitert werden. Für davon abgeleitete Operatoren wie z. B. *|=* ergab sich die Anpassung automatisch. Die Operationen sind zweckmäßig so organisiert, dass sie über Tabellen stattfinden. Die Tabelle 5.3 zeigt beispielhaft die Umsetzung der AND-Funktion. Die Werte in der ersten Zeile und in der ersten Spalte stehen für die Signale, die an den beiden Eingängen anliegen. Vollständige Listings der Operationstabellen befinden sich im Anhang A.1.10.

	0	1	Z	X	B	A	R
0	0	0	0	0	0	0	0
1	0	1	X	X	0	1	X
Z	0	X	X	X	0	X	X
X	0	X	X	X	0	X	X
B	0	0	0	0	0	0	0
A	0	1	X	X	0	1	X
R	0	X	X	X	0	X	R

Tabelle 5.3: AND-Funktionstabelle für Simulationskommandos

Je nach Datentyp lassen sich logische Werte unterschiedlich darstellen. Damit die verschiedenen Auslegungen ineinander überführbar bleiben, ist eine Typumwandlung (*type casting*) unumgänglich. So findet man für eine logische 0 verschiedene Interpretationen, z. B. als Charakter-Zeichen '0', als Ziffer 0 oder als SystemC-Wert `sc_logic_0`. Intern sind die logischen Werte als Aufzählungstyp (*enumeration type*) implementiert, wobei für die logische 0 mit der Null begonnen wird. Um nun ein Charakterzeichen in eine Aufzählungsvariable zu überführen, existiert eine *Konvertierungstabelle*, die für die neuen Logikwerte erweitert wurde.

Die Fehlerinjektion selbst geht am einfachsten mit der in der Abbildung 5.17 (b) vorgeschlagenen Variante. Das Listing 5.11 zeigt einen Auszug aus einer Simulationssteuerung mit Fehlerinjektion.

```

. . .
2  sc_initialize();
   CLK = 0; sc_cycle(1);
4  CLK = 1; sc_cycle(1);
   for (int I = 0; i < 1024; i++) {
6      CLK = 1; sc_cycle(10);
      CLK = 0; sc_cycle(10);
8  }
   M2.A6.write(sc_logic_A);
10 if (debug) cout << "sab1_A6" << endl;
   for (int I = 0; i < 1024; i++) {
12     CLK = 1; sc_cycle(10);
     CLK = 0; sc_cycle(10);
14 }
   M2.A6.write(sc_logic_R);
16 if (debug) cout << "sab1_A6 released" << endl;
   for (int I = 0; i < 1024; i++) {
18     CLK = 1; sc_cycle(10);
. . .

```

Listing 5.11: Simulations- und Fehlerinjektionssteuerung in der main.cpp

Am Anfang der Sequenz (bis zur Zeile 4) wird die Initialisierung der Schaltung vorgenommen. Von Zeile 5 bis 8 erfolgt eine normale Simulation. Durch den Gebrauch von `sc_cycle()` obliegt die Steuerung der Taktform (CLK) dem Anwender. Durch den Gebrauch einer Schleife (Zeile) wird dies vereinfacht. In Zeile 9 erfolgt die Injektion eines stuck-at-1-Fehlers in das Signal A6 in der Komponente und in der Zeile 15 ist die Fehlerinjektion wieder aufgehoben. Für viele Fehlerinjektionen kann die Ausarbeitung der Simulationssteuerung recht aufwändig sein. Daher wurde sie durch den Gebrauch von Skripten [184], ähnlich wie schon in Abschnitt C4.1.1 beschrieben, automatisiert. Eine Eigenschaft dieser Technik besteht darin – es geschieht keine Modifikation des DUT und dadurch wird auch das Berechnungsmodell nicht vergrößert.

### 5.4.3.2 Simulationskommandos mit Variablen

Variablen sind in den meisten Fällen innerhalb einer Komponente platziert. Die Fehlerinjektion beeinflusst daher typischerweise eine Eigenschaft eines Moduls, abgesehen von wenigen global wirkenden Variablen. Eine einfache Veränderung kann wie bei den Simulationskommandos mit Signalen (→5.4.3.1) durch eine Anweisung auch außerhalb des Moduls erfolgen. Die Syntax lautet:

```
Modulname.Variablenbezeichnung = Wert;
```

Ähnlich wie bei den Signalen ist die Injektion nur vorübergehend. Ein nächster Schreibvorgang hebt sie wieder auf. Da eine Variable mittels Speicher abgebildet wird, bleibt solch eine Injektion auch bis zum nächsten Schreibvorgang erhalten.

Für das Auftreten transienter Fehler, wie für Bit-Flip-Fehler, ist daher diese Form der Fehlererzeugung recht realistisch.

Eine Besonderheit haben Variablen in Hardwarebeschreibungssprachen zusätzlich. Für VHDL wurde in [132] von zwei Verhaltensformen der Fehlerinjektion bei Variablen berichtet. Zum einen gibt es temporale Variablen, die zu jedem Zeitpunkt beeinflussbar sind, auch wenn der Prozess gerade nicht aktiv ist. Zum anderen gibt es Variablen, die ein atemporales Verhalten aufweisen. Deren Inhalt ist über *WAIT*-Anweisungen hinaus nicht manipulierbar. Um in diesen Variablen eine Fehlerinjektion durchzuführen, wurde in [132] und [144] die Breakpoint-Technik vorgeschlagen. Hierbei wird beim Eintritt in den Prozess mit der zu manipulierenden Variable ein Haltepunkt (Breakpoint) gesetzt. Beim Eintreffen an diesem Haltepunkt wird dann erst der Variableninhalt gesetzt und anschließend weitersimuliert.

Ein ähnliches Verhalten von Variablen gibt es auch bei SystemC-Modellen. Einerseits gibt es Variablen, die global oder im Kopf eines Prozesses deklariert sind, diese haben ein temporales Verhalten. Andererseits gibt es Variablen, die erst in der Definition eines Prozesses deklariert sind. Diese weisen ein atemporales Verhalten auf, da deren Inhalt bei Prozessende an Gültigkeit verliert. Eine Abhilfe wäre es, jene Variablen schon im Modulkopf bekannt zu machen. Andererseits gibt es auch für C++



Entwicklungsumgebungen mit Debugger, die Haltepunkte erlauben [185]. Diese stehen somit auch für SystemC zur Verfügung, so dass auch hier die Breakpoint-Technik verwendet werden kann.

Die Injektion permanenter Fehler oder transienter Fehler mit deterministischer Dauer in Variablen ist in SystemC von vornherein schwierig. Es können jedoch in der Realität z. B. permanente Fehler durch Alterungsprozesse entstehen. Um solche Fehlerinjektionen in SystemC zu ermöglichen, wurden hierfür eigens neue Datentypen erschaffen. Die Erweiterung schon vorhandener Datentypen ist wenig sinnvoll, da C++ keinen Eingriff in seine proprietären Datentypen erlaubt.

Das Verhalten des neuen Datentyps ist so geprägt, dass es in Abhängigkeit eines Zugriff-Flags *lock* einen Schreibvorgang zulässt oder unterbindet. Im Listing 5.12 ist ein Ausschnitt aus der neuen Datentyp-Klasse *MyInt* dargestellt. Ein ausführlicheres Listing findet man im Anhang A.1.11.

---

```
2  class MyInt {
    private:
4      int dat;
      bool lock;
6  public:
      MyInt (int dat_=0) {                               //Konstruktor
8          this->dat = dat_;
          lock = false;
10     }
    . . .
12     inline MyInt& operator = (const MyInt& rhs) {
        if (!lock) this->dat = rhs.dat;
14         else {...}
        return *this;
16     }

18     inline bool setlock() {
        this->lock=true;
20         return true;
        }
22
        inline bool unsetlock() {
24         this->lock=false;
        return false;
26     }

28     inline bool getlock () {
        return this->lock;
30     }
    . . .
32 };
    . . .
```

---

Listing 5.12: Selbstdefinierter Datentyp zur Fehlerinjektion

In dieser Klasse wird der *Integer*-Datentyp um die Möglichkeit der Fehlermodellierung erweitert. Im SystemC-Modell ist dann bei der Deklaration der Datentyp *int* einfach gegen den neuen Typ *MyInt* auszutauschen. Die entsprechenden Werte werden in den privaten Member-Variablen *dat* und *lock* aufgenommen. Der eigentliche Integer-Wert ist in *dat* abgelegt und nur in den überladenen Operatoren wie +, -, =, usw. zu finden. Eine Basis-Operation ist somit nur für den Integerwert *dat* wirksam. Hieraus ergeben sich zum einen für den Anwender keine Inkonsistenzen beim Gebrauch des neuen Datentyps. Er verhält sich nach außen im Normalbetrieb wie der C++-Integer-Typ. Andererseits kann so eine aktive Fehlerinjektion durch eine Operation nicht ungewollt in eine andere Variable kopiert oder aufgehoben werden. Um ein genaues Debuggen mittels einer Signalkurven-Datei zu erreichen, wurden zusätzliche Methoden hinzugefügt. So kam z. B. die Funktion *sc\_trace()* hinzu, die das Aufzeichnen beider Member-Variablen realisiert.

Für den Zugriff auf das Zugriffs-Flag *lock* wurden drei separate Methoden implementiert :

- *setlock()* friert den aktuellen Stand der Variable ein. Es darf nur noch gelesen, aber nicht mehr geschrieben werden.
- *unsetlock()* hebt eine Fehlerinjektion wieder auf. Es darf sowohl gelesen als auch geschrieben werden. Bei der Erstellung einer Variablen ist dies die Voreinstellung.
- *getlock()* gibt Auskunft über den aktuellen Status der Fehlerinjektion.

Die Steuerung der Simulation und Fehlerinjektion ist ähnlich der, die im Listing 5.11 vorgeschlagen wurde. Um einen definierten Fehlerwert zu setzen, wird sequenziell zunächst der Fehlerwert zugewiesen und anschließend die Variable mit *Variablenbezeichnung.setlock()* gesperrt. Für ein feineres Fehlermodell kann es nötig sein, dass nicht der gesamte Variableninhalt sondern nur eine Bitstelle von einer Fehlerinjektion betroffen ist. In diesem Fall hilft die Einführung einer zusätzlichen Masken-Membervariable mit der spezifizierte Bits selektiert werden. Die Vorgehensweise der Implementierung ist ähnlich der oben vorgeschlagenen Methode.

Die gezeigte erweiterte Methode der Fehlerinjektion in Variablen kommt mit wenigen Änderungen im DUT aus. Einzig die Datentypen müssen bei der Deklaration ausgetauscht werden. Ein weiterer Vorteil besteht darin, dass man nun mit SystemC und den Simulationskommandos ausgefeiltere Fehlerinjektionen für eine Hardwarebeschreibung auch in Variablen vornehmen kann.

### 5.4.3.3 Simulationskommandos kombiniert mit anderen Techniken

Es ist grundsätzlich möglich, die verschiedenen Techniken der Fehlerinjektion (Saboteure, Mutanten, Simulationskommandos) in einem Schaltungsmodell zu kombinieren. Die Simulationskommandos erlauben das Setzen von Signalen und Variablen direkt im Schaltungsmodell. Saboteure und Mutanten sind nach der DUT-Modifikation Bestandteil der Modellbeschreibung. Es lassen sich die Vorteile der verschiedenen Techniken kombinieren. Die genauere Modellierung von Fehlern bei Saboteuren (z. B. Brückenfehler) oder von Mutanten (z. B. Austausch der Funktion) geht mit der nicht notwendigen Zusatzverdrahtung der Simulationskommandos einher. Kernidee ist es, dass die Ansteuerung der Fehlerinjektionen in den Saboteuren und Mutanten nicht mehr über zusätzliche Ports und Signale erfolgt, sondern durch Simulationskommandos ausgelöst wird. Für die Mutant-Technik ergibt sich dadurch eine Besonderheit. Mit dem Gebrauch einer vorhandenen Mutanten-Bibliothek müssen in einem Projekt nur die betreffenden Komponenten ausgetauscht werden, eine Änderung der Schaltungsstruktur ist nicht nötig.

Das Listing 5.13 zeigt die Umsetzung beispielhaft an einem Saboteur. Anstatt eines Fehlerinjektionsports *sa* ist dieser als ein internes Signal deklariert. Damit dieses Signal keinen unerwünschten Wert beim Simulationsaufruf annimmt, muss es im Konstruktor initialisiert werden (Zeile 9). Ansonsten erfolgt die Implementierung der Funktionalität genauso wie im Abschnitt C4.1 beschrieben.

---

```
2  . . .
   SC_MODULE(sasab) {
   . . .
4    sc_in<sc_logic> in;
    sc_signal<int> sa;
6    int c;
    . . .
8    sensitive << ... << sa;
        sa.write(0);
10   . . .
    void sasab::inject() {
12    c = sa.read();
        switch(c) {
14        case 0: . . .
        . . .
    }
```

---

Listing 5.13: Saboteur mit Simulationskommando

In [132] wurde Ähnliches in Verbindung mit dem neueren VHDL'93-Standard [192] vorgeschlagen.

Eine solche kombinierte Fehlerinjektion wird dort unter der Zuhilfenahme von globalen Signalen oder gemeinsamen globalen Variablen erzielt. Mit der hier vorgeschlagenen Methode in SystemC ist die Einführung solcher zusätzlicher Signale oder Variablen unnötig. Der Vorbereitungsaufwand zur Simulation fällt daher geringer aus.

### 5.4.4 Simulationsvergleich der Fehlerinjektions-Techniken

Ausschlaggebender Punkt für die Wahl einer Fehlerinjektionstechnik wird in erster Linie der zu modellierende Fehler sein. Bezüglich der Flexibilität und Vielseitigkeit ist die Mutant-Methode am stärksten, denn mit ihr lassen sich beispielsweise innerhalb eines Moduls Signale, Variablen und Funktionen manipulieren. Bezüglich eines geringen Simulationsaufwandes ist die Technik der Simulationskommandos von Vorteil. Es gibt auch Fehler, die sich mit allen drei vorgestellten Injektionstechniken realisieren lassen. Für das klassische Haftfehlermodell wurde dies anhand einiger Benchmark-Schaltungen näher untersucht. Dabei ist die Schaltung *c17* kleiner als *MSAB* und *MSAB* ist wiederum kleiner als die *c432*. In der Tabelle 5.4 sind die Simulationsergebnisse zu sehen. Die Simulationszeit ist im Verhältnis zur Simulationszeit mit den Simulationskommandos angegeben.

Injektionstechnik	Simulationszeit (normiert)		
	c17	MSAB	c432
Simulationskommandos	1,00	1,00	1,00
Mutanten mit Simulationskommandos	1,01	1,01	1,23
Mutanten	1,05	1,36	1,96
Saboteure mit Simulationskommandos	1,28	1,32	1,81
Saboteure	1,30	1,40	1,84
FIT (VHDL Fault Injection Tool)	3,71	3,14	5,21

Tabelle 5.4: Simulationsergebnisse für verschiedene Injektionstechniken

Die Technik der Simulationskommandos ist hinsichtlich der Ausführungszeit am besten. Auch die Kombination der Mutant-Technik mit den Simulationskommandos ist schneller als die reine Mutant-Technik. Gleiches gilt für die Saboteure. Zusätzlich sind zum Vergleich Simulationen mit dem FIT-Tool dargestellt. Beim FIT handelt es sich um ein Fehlerinjektionstool für strukturelle VHDL-Beschreibungen [229]. Die Simulationsergebnisse sind mit diesem Programm deutlich am schlechtesten.

## 6 Mixed-Language-Co-Simulation

In diesem Kapitel wird eine Kombination präsentiert, bestehend aus einer SystemC- und einer VHDL-Simulation. Für die VHDL-Simulation wird hierzu das FIT [229] verwendet. Der vorgestellte Lösungsweg ist auch dazu geeignet, eigene Applikationen mit einer SystemC-Simulation zu koppeln.

Das Ziel einer *Mixed Simulation* (dt. *gemischte Simulation*) oder einer *Co-Simulation* (dt. *kooperierende Simulation*) ist es, ein heterogenes System in einer Umgebung und in einem gemeinsamen Rahmen endlicher Zeit zu simulieren. Bei einer Mixed Simulation kann es verschiedene Gründe für eine Mischung geben. Es lassen sich z. B. verschiedene Signaltypen (*signal mixed mode*), unterschiedliche Beschreibungssprachen (*multi-language, mixed-language*) [207] oder mehrere Abstraktionsebenen (*multi-level*) gemischt simulieren. Es existieren auch Simulatoren, die einen Mix mehrerer Sachverhalte erlauben [203]. Eine Co-Simulation geht etwas weiter und meint typischerweise die Simulation in mindestens zwei unterschiedlichen Domänen. So können zwei unabhängige Simulatoren miteinander gekoppelt sein oder HW und SW, mit ganz unterschiedlichen Aufgabenbereichen, in einer Umgebung simuliert werden.

In dieser Arbeit wird die vorgestellte Anwendung als *Mixed-Language-Co-Simulation* bezeichnet. Zum einen liegt es begründet in der Nutzung von Charakteristiken beider Ansätze. Zum anderen wird die englischsprachige Bezeichnung verwendet, da bisher ein deutschsprachiges Pseudonym in der Fachwelt unüblich ist.

Für die gemeinsame Simulation eines Systems mit Komponenten in unterschiedlichen Beschreibungssprachen (*mixed-language*) gibt es eine Reihe von Gründen. Im Folgenden sind einige wichtige Motive aufgeführt:

- **Arbeitsteilung:** Verschiedene Teams arbeiten an unterschiedlichen Teilen des Systems mit unterschiedlichen Sprachen. Z. B. arbeitet für eine Anwendung eine Arbeitsgruppe an der Hardware des Moduls und nutzt hierfür eine HW-Beschreibungssprache und eine andere Gruppe beschäftigt sich mit der Software und verwendet C++-Code.
- **Hierarchienutzung:** Ein Teil der Schaltung wird auf einer höheren Ebene der Abstraktion beschrieben. Ein Vorteil ist zumeist eine signifikante Erhöhung der Simulationsgeschwindigkeit. Zum Beispiel werden dabei gern abstrakte "C"-Modelle mit Verilog/VHDL gemischt.
- **Heterogenität:** Die einzelnen Teile des Systems sind derart unterschiedlich, dass die Beschreibung mit einem einzigen Mittel nicht ausreichend ist. Bei einer Mixed-Signal-Schaltung arbeitet z. B. eine Arbeitsgruppe am Analogteil mit

einem grafischen Tool, welches eine Netzliste liefert und das Team der digitalen Seite verwendet Verilog. Oft wird auch die Notwendigkeit zitiert, VHDL und Verilog zu mischen [199,204]. Hier wird die genauere Verhaltensnachbildung von Verilog mit der universelleren Systembeschreibung von VHDL gemischt.

- **Testbench-Erzeugung:** Die für eine Simulation nötige Zuführung von Eingangsmustern und das Aufzeichnen der Ergebnisse erfolgt mit einer anderen Programmiersprache, als die Beschreibungssprache in der das Modell abgebildet ist. Zum Beispiel wird für die Testbench SystemC/SCV, wo u. U. ein constraint randomization zum Einsatz kommt [41], verwendet und für die Schaltung wird VHDL genutzt.
- **Wiederverwendung:** Modelle aus früheren Projekten werden wieder verwendet (*Re-using*). Zum Beispiel war ein früheres Modell in einem anderen Format als das im Team aktuell verwendete Beschreibungsmittel. Um sich den Aufwand der Überführung zu sparen, wird eine Mixed Simulation genutzt.
- **IP-Einbindung:** Das Teil-Modell eines Systems wird von einem Anbieter dazugekauft. Eine IP-Komponente liegt vorzugsweise in einer Form vor, die einen Know-how-Schutz für den Lieferanten gewährleistet. Dies kann z. B. für eine Simulation ein Modell sein, welches als Bibliothek bzw. ausführbares Objekt (DLL) für die Simulation geliefert wird.
- **Verifikation oder Validierung:** Zwei gleichartige Systeme in unterschiedlichen Beschreibungssprachen oder auf unterschiedlichen Ebenen des Entwurfs werden simultan simuliert und ihre Ergebnisse miteinander verglichen [199]. So lässt sich z. B. bei einer Verfeinerung des Entwurfprozesses die Validierung der Modelle erzielen.
- **Parallelisierung:** Das Vorhandensein zweier Programme bei einer Co-Simulation macht sie für ein verteiltes Rechnen interessant [204]. Allerdings gilt es dabei, das Problem der Synchronisation zu lösen. Vorteile können eine kürzere Ausführungszeit und ein größerer zur Verfügung stehender Speicherraum sein.

Beim traditionellen Hardware-Entwurf genügt der Umgang mit einem Beschreibungsmittel bzw. mit einem Werkzeug. Durch das Wesen des SoC-Entwurfs, wie dem Vorhandensein unterschiedlicher Komponenten, wird die Mixed Simulation und auch die Mixed Language Simulation zunehmend wichtiger.

Üblicherweise wird bei der gemischten Simulation in zwei Kategorien unterschieden, in die Co-Simulation und in die Mixed-Language-Kernel-Simulation. Jeder Weg hat dabei seine Vor- und Nachteile.

Die **Co-Simulation** ist bereits länger bekannt und zeichnet sich dadurch aus, dass sie mehrere unabhängige Simulator-Kernel beinhaltet. Der Austausch erfolgt über eine Kommunikationsinterface, welches eine Bibliothek oder selbst wieder eine separate Anwendung ist. Für eine Co-Simulation sind einige Mechanismen bekannt:

- 
- *Foreign Language Interface (FLI)*: ist ein Interface, welches in VHDL definiert ist. Es erlaubt den Aufruf von Funktionen in C und damit die Anbindung anderer Anwendungen an den Simulator. Es hat keinen speziellen Mechanismus, um mit SystemC zu kooperieren, aber in [206] findet man eine Umsetzung für ein SystemC-Modell. Insbesondere durch die Implementierung in Modelsim [24] erlangte dieses Konzept an Bekanntheit.
  - *Open Model Interface (OMI)*: Dieses Konzept ist nach IEEE 1499 [192] standardisiert. Es war für einen problemlosen Austausch von IPs gedacht. Jedoch sind Modelle und entsprechende Anwendungen schlecht verfügbar bzw. arbeitet es unzuverlässig [200].
  - *Programming Language Interface (PLI)*: Dieses Interface ist in der IEEE 1364 standardisiert und für Verilog vorgesehen. Über C-Prozeduren erlaubt es einen Zugriff auf simulatorinterne Strukturen und Funktionen. In den meisten Fällen ist der Umgang mit diesem Interface recht umständlich, trotz nützlicher Wizard-Funktionen in modernen HDL-Simulatoren [25]. Eine erste gekoppelte Anwendung von SystemC und Verilog mit der PLI-Schnittstelle findet man in [206]. Die letzte Generation der PLI-Versionen ist das *Verilog Procedural Interface (VPI)*, welches im IEEE Standard 1364-2001 definiert ist.
  - *VHDL Procedural Interface (VHPI)*: Diese Schnittstelle ist ähnlich der VPI, jedoch für VHDL vorgesehen. Innerhalb des Standards IEEE 1076 für VHDL findet dieses Interface seine Definition [208,209]. Für den Umgang und die Schwierigkeiten gilt das Gleiche wie für das PLI-Interface.
  - *Proprietäre Mechanismen*: Unter diesem Punkt sind alle Interface-Implementierungen zusammengefasst, die produktabhängig sind. In den meisten Fällen sind die Lösungen ausgereift, aber auf die jeweilige Anwendung beschränkt.

Beim Einsatz eines Interface-Konzeptes gilt es zu beachten, dass die gewünschte Funktionalität auch vom gewählten Programm unterstützt wird. Selten sind alle im Interface-Standard aufgeführten Funktionen in einem Werkzeug integriert. Eine wesentliche Eigenschaft der Co-Simulation ist, dass sie zusätzlichen Overhead für den Austausch von Daten zwischen den Simulations-Kernen erzeugt. Dieser Mehraufwand äußert sich nicht nur durch zusätzlichen Code in der Schnittstellenbeschreibung, sondern auch in zusätzlicher Simulationszeit. Besonders bei PLI/VPI und FLI/VHPI bildet dies einen Flaschenhals in der Simulation [24,25].

Die **Mixed-Language-Kernel-Simulation** verfolgt den Ansatz, die komplette Simulation mit einem Programm bzw. einem Simulations-Kernel durchzuführen. Sie wird deshalb zuweilen als *Single-Kernel-Simulation* bezeichnet. Die verschiedenen Modellbeschreibungen werden in einen gemeinsamen *native code* überführt (→3.1.3) und dieser dann vom Simulator ausgeführt. Für den Anwender ergibt sich der Vorteil,

nur in einer Programmumgebung zu hantieren. Bei einer Simulation mit C/C++-Code wird bei einigen Anwendungen der kompilierte Code nicht in den vom Simulator auszuführenden *native code* übersetzt, sondern er wird in einen direkt von der CPU ausführenden Maschinencode kompiliert [26]. Dieser liegt meist in Form von Bibliotheksmodulen vor. Die Kopplung der unterschiedlichen Beschreibungen sieht so aus, dass ein Submodul in einer Sprache per Deklaration in einer höheren Ebene einer anderen Sprache bekannt gemacht und instanziiert wird. Raffinierte Simulatoren erlauben sogar die mehrfache Schachtelung von unterschiedlichen Beschreibungen. So ist es möglich, eine VHDL-Beschreibung zu definieren, die auf ein Verilog-Modul verweist und dieses Verilog-Modul besitzt wiederum ein VHDL-Submodul.

Ein wesentlicher Vorteil der Mixed-Language-Kernel-Simulation ist der geringe Kommunikations-Overhead. Damit erhält man eine deutlich höhere Simulationsgeschwindigkeit im Vergleich zur Co-Simulation. Ein Nachteil an dieser Simulationsform ist, dass nur ein Teil des definierten Sprachumfangs unterstützt wird. So gibt es oft Einschränkungen hinsichtlich der zu verwendenden Datentypen oder nur eine teilweise Unterstützung zusätzlicher Bibliotheksfunktionen, wie sie z. B. bei SystemC mit der SCV existiert. Bei nahezu allen Werkzeugen muss der Anwender an seinem Modell Änderungen vornehmen oder Konventionen einhalten, damit das Programm eine Mixed-Language-Kernel-Simulation durchführt.

Im verbleibenden Teil des Kapitels wird zunächst auf den verwendeten VHDL-Simulator FIT eingegangen. Anschließend ist das Konzept der eigenen Umsetzung zu sehen. Dem schließen sich eine Erläuterung der vorgenommenen Änderungen in der SystemC-Bibliothek und der Aufbau des Simulationsprojektes an. Vergleichende Simulationsergebnisse, die die Effizienz demonstrieren, schließen das Kapitel ab.

## 6.1 Das Fault Injection Tool

Um sequenzielle Schaltungen, wie sie z. B. Prozessorkomponenten darstellen, auf ihr Verhalten bezüglich des Auftretens von Fehlern zu simulieren, entstand an der BTU Cottbus der Fehlersimulator HEARTLESS [210]. Aus diesem HEARTLESS-Projekt ging das leistungsfähigere VHDL-Werkzeug FIT (Fault Injection Tool) hervor [229,33]. Traditionelle Werkzeuge für Hardwarebeschreibungen beherrschen die Aufgaben der Logiksimulation recht gut. Ist es aber erforderlich eine Simulation mit zusätzlichen Schaltungsfehlern auszuführen, werden manuelle Eingriffe und Aufwändungen erforderlich. Um diesen Zusatzaufwand zu vermeiden, entstand die Motivation zum FIT-Projekt. Neben der ursprünglichen Logik-Simulation erlaubt dieses Programm eine komfortable Fehlerinjektion. Ausgeführt ist FIT als interpretierender, ereignisgesteuerter Gate-Level-Simulator (→3.1). Bei Bedarf lassen sich abstrakte



VHDL-Funktionen als Makros definieren und in FIT integrieren. Darüber hinaus besitzt das FIT noch weitere wichtige Eigenschaften:

- **Hierarchien:** Das Programm ist in der Lage, Strukturbeschreibungen mit nahezu beliebig vielen Ebenen zu verarbeiten.
- **High-Level-Description (HLD):** Das Programm ist in der Lage, funktionale Module zu simulieren, die in einer Hochsprache wie C++ [177] oder Delphi [186] erstellt wurden. Zur Anbindung an den Simulator besitzt dieser ein C-Interface. Die HLD selbst liegt in Form einer kompilierten Bibliothek vor. Die Vorteile einer HLD liegen in einer einfacheren und schnelleren Modellierung, da das Verhalten mit Hochsprachen-Befehlen erzeugt werden kann und in einer oftmals schnelleren Simulation. Es lassen sich auch schon vorab Module simulieren, für die noch keine Implementierung der Schaltungsstruktur existiert.
- **Schaltungsmodelle:** Es lassen sich kombinatorische als auch sequenzielle Schaltungen simulieren. Bei den synchronen sequenziellen Modellen sind neben einer Einfachtaktung auch Mehrfachtaktungen erlaubt.
- **Fehlerinjektion:** Das Werkzeug gestattet neben der Injektion klassischer Haftfehler auch das Einfügen von transienten und Bit-Flip-Fehlern. Optional kann man diese Fehler mit einer Verzögerung beaufschlagen. Neben der Einzelfehlersimulation (SPSF), wie sie für die ATPG gebräuchlich ist, existiert die Möglichkeit, auch Mehrfachfehler zu injizieren, wie sie für die Verlässlichkeitsanalyse (→5.1.1) sinnvoll ist.
- **Add-ons:** Neben dem reinen Simulator existieren verschiedene Zusatzwerkzeuge, die dem Anwender die Arbeit erleichtern. Hierzu zählt ein Fault-Collapsing-Tool [32], welches die Ausführungszeit einer Simulation deutlich reduzieren kann. Dies geschieht dadurch, dass sich anhand von Äquivalenz- und Dominanzverhalten an Komponenten die Anzahl der notwendigen, zu injizierenden Fehler reduzieren lässt [23]. Ebenso stehen Programme bereit, die beispielsweise die Generierung von Eingabemustern und der Fehlerlisten übernehmen.
- **ANSI C++:** Das Programm ist in ANSI C++ verfasst. Auf dieser Basis lässt es die Möglichkeit offen, es auf verschiedenen Plattformen zu portieren. Es stehen derzeit Ausführungen für MS Windows und für Linux zur Verfügung.

Die Abbildung 6.1 zeigt die Organisation eines FIT-Projektes. Kern der Simulation bildet der FIT-Simulator.

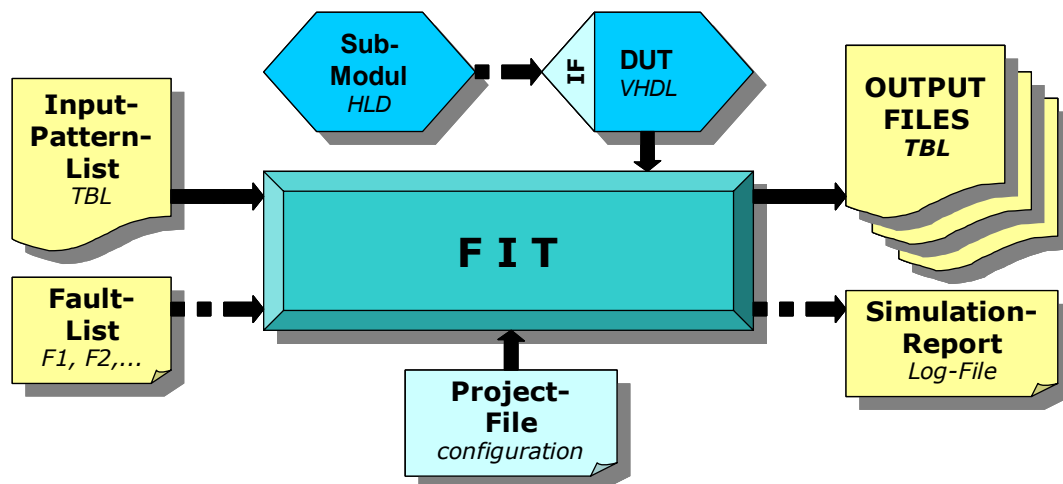


Abbildung 6.1: FIT-Projekt

Das FIT wird mit einer Reihe von Eingaben versorgt und erzeugt zweckmäßigerweise während der Simulation verschiedene Ausgaben. Die Ein- und Ausgaben die in der Abbildung 6.1 mit einem gestrichelten Pfeil dargestellt sind, haben für die Ausführung eine optionale Bedeutung. Grundlage der Simulation ist die Schaltungsbeschreibung, welche im VHDL-Format oder in einem proprietären Binärformat vorliegen muss. Der Vorteil des Binärformats ist ein schnelles Einlesen der Beschreibung. Zusätzlich können abstrakte Module als dynamisch gelinkte Bibliothek, unter Windows sind es DLL-Dateien [217] bzw. unter Linux Shared-Object-Dateien, über die VHDL-Beschreibung in die Simulation mit eingebunden werden. Hierzu existiert eine spezielle Interface-Konvention (→6.4). Die Eingaben für die Schaltungseingänge werden in einer Waveform-Datei in Alteras Table-File-Format (.tbl) abgelegt [211]. Gleichmaßen werden die Simulationsausgaben in eine TBL-Datei geschrieben. Für eine Simulation mit Fehlern benötigt man noch eine Fehlerliste, die als eine Datei im Textformat vorliegt. Bei der Einzelfehlersimulation wird für jeden Fehler der komplette Satz an Eingangsmuster simuliert. Dabei erfolgt für jeden Fehler die Ausgabe in einer separaten Datei. Bei Mehrfachfehlern wird für jede Fehlergruppe äquivalent vorgegangen. Zusätzlich lassen sich weitere Ausgaben, die den Verlauf der Simulation dokumentieren, in einer Report-Anzeige schreiben. Hierzu existieren verschiedene Debug-Modi.

Die Steuerung der Simulation erfolgt über eine Konfigurationsdatei. In der Konfiguration lassen sich die Namen der zu lesenden und zu schreibenden Dateien, der Simulationsmodus (ob mit Fehler), etc. festlegen.

## 6.2 Konzept der Mixed-Language-Co-Simulation

Die Mixed- bzw. Co-Simulation von HDLs, insbesondere die von VHDL und Verilog, ist nicht neu und in unzähligen Arbeiten umgesetzt. Ebenso ist es bei allen namhaften Herstellern von HDL-Simulatoren ein Muss, eine Mixed Simulation mit SystemC einzuräumen [25,24,28]. Es gibt allerdings deutlich weniger Arbeiten, die sich mit der Simulation von Fehlern in einer gemischten Umgebung beschäftigen [205]. Noch seltener sind Veröffentlichungen bekannt, die von einer Mixed-Language- oder Co-Simulation in Bezug auf SystemC und Fehlerinjektion handeln [38].

Die hier vorgestellte Lösung sieht so aus, dass zwei Simulationskernel existieren, anders also als bei Mixed-Language-Single-Kernel-Simulation. Jedoch arbeiten sie nicht so unabhängig voneinander wie bei einer Co-Simulation. Während der VHDL-Simulator FIT unverändert bleibt, werden an der SystemC-Umgebung Modifikationen vorgenommen. Das Ergebnis der Modifikationen ist ein SystemC-Modell, welches als dynamische Link-Bibliothek vorliegt, ähnlich wie bei einigen Mixed-Language-Simulatoren mit einem Single-Kernel [25]. Allerdings wird der SystemC-Simulations-Kernel nicht entfernt, sondern nur modifiziert und beibehalten. Dabei wurde der Kernel als ausführbarer Thread implementiert. Das SystemC-Modell kann dadurch nicht mehr eigenständig ausgeführt werden. Es lässt sich jedoch in die Simulationsumgebung vom FIT einbinden und der VHDL-Simulator steuert als Master den Fortschritt der SystemC-Simulation. Da hierbei Ähnlichkeiten sowohl zur Mixed-Language-Single-Kernel-Simulation als auch zur Co-Simulation bestehen, wurde der Begriff einer *Mixed-Language-Co-Simulation* für das in dieser Arbeit vorgestellte Verfahren eingeführt.

## 6.3 Modifikationen an der SystemC-Bibliothek

Auf den ersten PC-Systemen war es üblich, mit z. B. MS DOS als Betriebssystem [212], dass man zu einem Zeitpunkt nur ein Programm ausführte. Standen mehrere Aufgaben zur Disposition, so mussten diese sequenziell ausgeführt werden. Heutige Rechnersysteme erlauben hingegen einen Mehrprogrammbetrieb [213]. Dabei sind mehrere Prozesse (Tasks) quasi-parallel aktiv. Die Verwaltung der Tasks übernimmt üblicherweise der Scheduler des Betriebssystems [213]. Ein Prozess ist hierbei nicht mehr ein Programm selbst wie bei den frühen Systemen, sondern eine Programminstanz. Eine Konsequenz davon ist beispielsweise, dass ein Programm in einer Sitzung mehrfach ausgeführt werden kann. Die Ausführung mehrerer Tasks in einer Sitzung ist eine wesentliche Voraussetzung für eine Co-Simulation. Damit es zwischen den Tasks zu keinen Ressourcen-Konflikten kommt, muss deren Verteilung vernünftig geregelt sein. Konkurrieren zwei Prozesse um eine Ressource, wie z. B. ein Register, muss der Registerinhalt für die aktuell aktive Task gesichert werden, bevor die CPU der nächsten Task zugeteilt wird. Bei einem realen Task-Wechsel in einem

typischen Betriebssystem betrifft dies mehr als nur ein Register. Hierbei werden Informationen zur Identifikation, zu Inhalten und Kontrollinformationen des Prozesses gesichert, was dem sogenannten Prozesskontext entspricht. Neben den Prozessor- und Statusregistern sind noch Adressräume (ganze Speicherinhalte) zu sichern. Solch ein Task- bzw. Kontextwechsel (engl. *context switch*) kostet viel Zeit, für gewöhnlich mindestens einige tausend CPU-Zyklen. Bei einer Schaltungssimulation besteht zumeist eine starke Abhängigkeit des Datenaustausches zwischen den Komponenten, d. h., bei einem Simulationszyklus ist die Wahrscheinlichkeit recht hoch, dass diese Komponenten auch häufiger ausgeführt werden. Insbesondere gilt dies für getaktete Module. Bei einer Co-Simulation wäre das System zu großen Zeitanteilen mit Kontextwechseln beschäftigt, was nicht sehr effizient ist.

Neben den Tasks als schwergewichtige Prozesse gibt es in modernen Rechnern die Möglichkeit von leichtgewichtigen Prozessen. Diese Prozesse werden als Threads bezeichnet. Threads werden gern dazu benutzt, um eine Parallelität von Routinen in einem Programm zu erreichen. Da bei Threads ein gemeinsamer Adressraum benutzt wird, muss man bei einem Kontextwechsel weit weniger Daten sichern als bei einem Taskwechsel. Threads teilen sich im gemeinsamen Adressraum die globalen Variablen. Diese Eigenschaft erleichtert auch den Datenaustausch der hier vorgestellten Mixed-Simulation. Bei einem Task-Wechsel sind nur die Prozessor-/Statusregister und einige wenige Zusatzinformationen zu sichern [214]. Aus diesem Grund benötigt ein Zustandswechsel bei einem Thread weit weniger Zeit als bei einer Task [216]. Folglich verkürzt dies auch die gesamte Simulationszeit. Eine wesentliche Voraussetzung zur Implementierung von Threads sind funktionale Unabhängigkeiten in einem Programm. Diese Notwendigkeit ist durch die Systemmodellierung mit zwei unterschiedlichen Hardwarebeschreibungsmitteln gegeben.

Bei den Threads unterscheidet man in zwei Kategorien. Zum einen gibt es die User-Level Threads (ULT) und zum anderen die Kernel-Level Threads (KLT). Bei den ULT übernimmt die Anwendung die Verwaltung der Threads, für das Betriebssystem sind diese nicht sichtbar. In SystemC gibt es auch diese ULTs. Hier sind es standardmäßig die SystemC-Prozesse (→2.4), welche als User-Level Threads arbeiten. Für neuere SystemC-Bibliotheken gibt es die Option, die Prozesse *sc\_thread* und *sc\_cthread* unter Linux auch als KLTs ablaufen zu lassen. Hier herrscht jedoch noch Uneinigkeit, inwiefern diese Variante den experimentellen Status verlässt und einen praktischen Nutzen erhält [29].

Viele Betriebssysteme implementieren die Threads in ihrem Systemkern. Konzeptionell ähneln sich die Implementierungen meistens, aber in ihrer Programmierschnittstelle und in ihren Einzelheiten unterscheiden sie sich zum Teil deutlich. Das macht Thread-verwendende Applikationen nicht gerade portabel. Um diesen Umstand abzuschaffen, entstanden die sogenannten POSIX-Threads oder auch Pthreads [215]. Diese Bestrebung wurde durch die IEEE-Organisation [192] im POSIX-Standard P1003.1c standardisiert. In der folgenden Implementierung der Mixed-Simulation wurden deshalb die POSIX-Threads verwendet. So lässt sich ein

Simulationsmodell mit wenigen Änderungen sowohl unter Linux als auch unter MS Windows ausführen.

Die wesentlichsten Änderungen in der SystemC-Bibliothek wurden in der *sc\_main.cpp* im Kernel-Bereich vorgenommen. Das Listing 6.1 zeigt einen wesentlichen Ausschnitt aus dieser Datei.

---

```
void *t_test(void *param)
2 {
    int status = 0;
4     try {
        //pln();
6     // hier stehen die früheren Anweisungen
        . . .
8     . . .
        return NULL;
10 }

12 int create_SystemC_Thread() {
    pthread_t thread;
14     int *param = 0;
        int make_Thread = pthread_create(&thread, NULL, t_test, param);
16     if(make_Thread != 0){
        printf("Thread was not created!\\n");
18     }
        pthread_detach(thread);
20     return(0);
}
```

---

Listing 6.1: Änderungen in der *sc\_main.cpp*

Normale Anwendungen, die in C++ geschrieben sind, erklären ihren Programmeintrittspunkt mit der *main()*-Funktion. Eine normale SystemC-Anwendung weicht hiervon ab und ersetzt diese Routine in der Anwendung durch die SystemC-spezifische *sc\_main()*-Funktion [39]. Der eigentliche Programmstart erfolgt dann indirekt über die in der *sc\_main.cpp* implementierte *main()*-Funktion, welche vom C++-Kompiler erwartet wird. Da das SystemC-Zielobjekt keine selbständige, ausführbare Anwendung ist, sondern nach der Fertigstellung als dynamische Link-Library vorliegt, kann dieser *main()*-Einstiegspunkt entfallen. In der *main()*-Funktion der SystemC-Bibliothek befinden sich einige Anweisungen, wie z. B. das Abfangen von Ausnahmefehlern. Um diese Funktionalität nicht zu verlieren, wurden Teile der *main()*-Funktion in einer Ersatzfunktion (Zeile 1) aufgenommen. Diese Funktion ist auch dafür verantwortlich, dass der SystemC-Simulations-Thread später in der *Elaboration*-Phase instanziiert wird. Außerdem kam noch eine Funktion hinzu, die für die Erzeugung eines Threads sorgt (Zeile 12). Damit diese auch außerhalb der Bibliothek zugreifbar ist, wurde sie nach außen sichtbar gemacht. Nach der Kompilierung steht eine statische Multithreaded-Bibliothek für den weiteren Gebrauch zur Verfügung.

## 6.4 Die Kopplung in einem Simulationsprojekt

Die Kopplung der beiden Simulationen von VHDL und SystemC erfolgt über ein spezielles Interface. Die Abbildung 6.2 zeigt einen Überblick von einem Mixed-Simulations-Projekt.

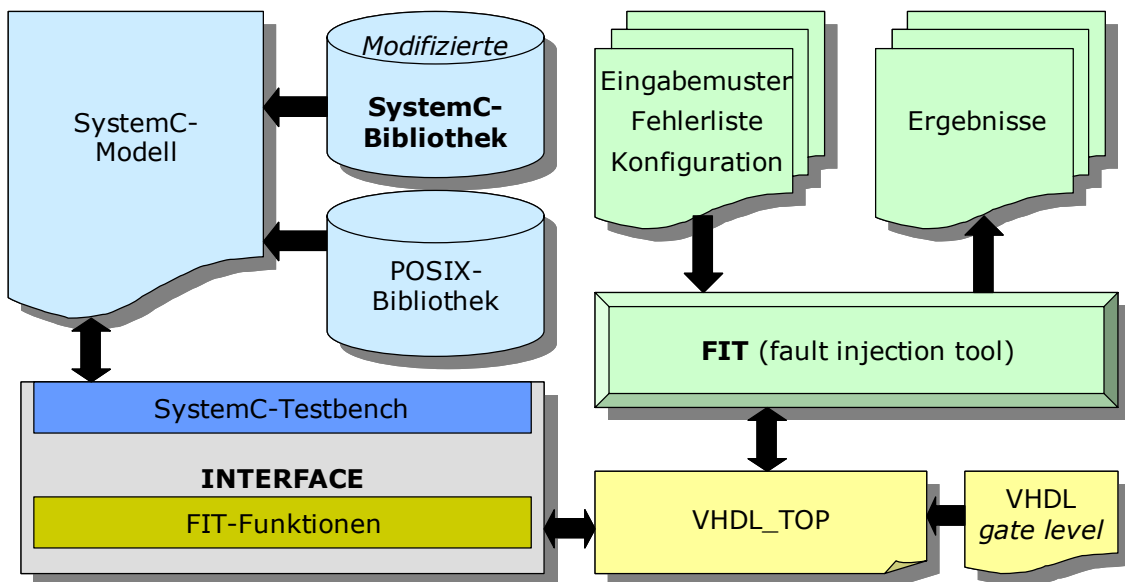


Abbildung 6.2: Mixed-Simulations-Projekt

Auf der rechten Seite der Abbildung findet man den VHDL-Teil der Simulation. Der Kern ist das FIT. Dieses Werkzeug liest aus drei Dateien die Muster für die Eingänge, die Liste mit den zu injizierenden Fehlern und die Konfiguration des Projektes. Während der laufenden Simulation werden die Ergebnisse abgelegt. Eine Schaltungsbeschreibung in VHDL kann in dieser Umgebung als Strukturbeschreibung implementiert sein. Es ist aber auch möglich, das komplette Modell als SystemC-Beschreibung mit dem VHDL-Fehlerinjektionstool auszuführen. In beiden Fällen ist hierfür eine VHDL-Top-Datei nötig. In dieser findet das FIT eine Deklaration der Ein- und Ausgänge per ENTITY-Block [189]. Für das FIT wurden spezielle Anweisungen eingeführt, die den Pragma-Anweisungen in C++ [177] ähneln. Mit deren Hilfe weiß das FIT, ob es eine Komponente als HLD oder als VHDL-Modell ausführen soll. Mit der Anweisung:

```
--USE HLD MULTIPLIER
```

im Kopf der VHDL-Top-Datei erfährt der Simulator beispielsweise, dass er eine HLD mit der Bezeichnung *MULTIPLIER* benutzen soll. Andere VHDL-Werkzeuge, die die gleiche Schaltungsbeschreibung simulieren wollen, stört dieser Befehl nicht, da er ihnen als Kommentar erscheint.

Auf der linken Seite befindet sich das SystemC-Projekt. An dem Schaltungsmodell in SystemC muss für die Mixed-Simulation keine Änderung erfolgen. Bei der Kompilation bzw. beim Linken müssen nur die POSIX-Bibliothek und die modifizierte SystemC-Bibliothek (→6.3) mit eingebunden werden.

Zur Kopplung und Kommunikation zwischen den beiden Beschreibungsformen existiert das Interface. Zur Anbindung an das SystemC-Modell beinhaltet es eine SystemC-Testbench, welche sich sonst üblicherweise in einer *main.cpp* befindet. Für die Kopplung mit dem FIT werden im Interface fünf Funktionen integriert. Eine HDL wird anhand eines *Identifiers* erkannt, welcher jeder Funktion/Prozedur übergeben wird, um den korrekten Speicherbereich anzusprechen. Die fünf Funktionen haben folgende Aufgaben:

- *initial(int identify)*: Diese Funktion erledigt die Reservierung und gegebenenfalls die Initialisierung des erforderlichen Speichers. Sie wird vor dem Simulieren aufgerufen.
- *destruct(int identify)*: Diese Routine gibt den reservierten Speicher wieder frei und wird am Ende der Simulation aufgerufen.
- *reset(int identify)*: Diese Funktion setzt die Speicherinhalte auf die Voreinstellungswerte. Sie ist beim Zurücksetzen der Simulation erforderlich.
- *calculate\_1(int identify, int NrOfInputs, int\* Inputs)*: Diese Routine bekommt neben dem Identifier noch die Anzahl der Eingänge und einen Pointer auf ein Feld mit Eingangswerten übergeben. Sie errechnet auf Basis der Eingangsdaten die Werte an den Ausgängen.
- *calculate\_2(int identify, int NrOfOutputs, int\* Outputs)*: Diese Prozedur bekommt neben dem Identifier noch die Anzahl der Ausgänge übergeben und verweist mit einem Pointer auf ein Feld mit Ausgangswerten. Sie übergibt die errechneten Ausgangswerte an den FIT-Simulator.

Das Listing 6.2 zeigt als Beispiel den *calculate\_1*-Teil aus der Interface-Beschreibung. In diesem Ausschnitt kann man sehen, wie vom FIT die SystemC-Simulation gestartet und synchronisiert wird. Bei einem ersten Start der *calculate\_1*-Funktion ist das *PassFlag* noch 0 (Zeile 12). In diesem Fall wird ein SystemC-Thread aus der modifizierten SystemC-Bibliothek erzeugt (Zeile 14). Damit weitere Funktionsaufrufe während der Simulation nicht noch weitere Threads erzeugen, wird das Flag am Ende des ersten Aufrufs auf 1 gesetzt.

Mit Hilfe des *GoOn*-Flags erfolgt die Synchronisation zwischen der SystemC-Simulation und dem FIT. Mit einer 1 bzw. einem true signalisiert die Routine, dass ihre Berechnung abgeschlossen ist. Für die Freigabe des Threads wird ein Trick mit dem *Sleep(0)*-Befehl benutzt. Normalerweise dient der *Sleep(Dauer)*-Befehl zur Ausführung einer Pause mit der übergebenen Zeitdauer. Ein Wert von 0 wäre demzufolge logisch nicht sinnvoll.

Aber ein Thread kann mit dieser Kernel-Funktion dem Scheduler mitteilen, dass er untätig ist. Der Scheduler entzieht ihm daraufhin die CPU und vergibt sie weiter.

---

```
2  extern "C"  char* calculate_1(int identify, int NrOfInputs, int*
    Inputs)
4  {
    int argc = 0;
6   char** argv = NULL;
    if(NrOfInputs != NoGI) {
8       return strdup( "Inputs are not correct" );
    }
10
    globalInputs = Inputs;
12   if(PassFlag == 0) {
        GoOn = 1;
14       create_SystemC_Thread();
        while(GoOn == 1 ) {
16           Sleep(0); }
    }
18   else {
        GoOn = 1;
20       while(GoOn == 1 ) {
            Sleep(0); }
22   }
    PassFlag = 1;
24   return 0;
}
```

---

Listing 6.2: calculate\_1-Routine im Interface-Teil

Alternativ kann anstatt des *Sleep(0)*-Befehls auf einer Intel CPU mit Hyper-Threading der neue Befehl *Pause* benutzt werden (in C++: `_asm PAUSE`). Ziel dieser Ersetzung ist ein deutlich schnellerer Umschaltvorgang zwischen den Threads, da hier die speziellen Eigenschaften der Hyper-Threading-Technologie genutzt werden [219]. In eigenen praktischen Versuchen waren solche Zeitvorteile kaum messbar. Andere Möglichkeiten der Synchronisation bieten viele Thread-Bibliotheken mit den in ihnen implementierten Synchronisationsfunktionen, wie z. B. den MUTEX-Routinen [215]. Da der Umschaltvorgang anscheinend weniger ins Gewicht fällt als die Berechnung, wie beim *Pause*-Befehl, wurde eine solche aufwändigere Lösung nicht näher betrachtet. Der Einsatz des *Sleep(0)*-Befehls hat weiterhin den Vorteil, dass der FIT-Simulator für die Mixed-Simulation nicht umgeschrieben und neu kompiliert werden muss.

Einen Teil eines Testbench-Beispiels vom Interface für das SystemC-Modell ist im Listing 6.3 dargestellt.



---

```

2      // Testbench für das SystemC-Modell
3
4      SC_MODULE (testbench) {
5          sc_out<sc_logic> A0_p, A1_p, B0_p, B1_p, CIN_p;
6          sc_in<sc_logic> SUM0_p, SUM1_p, COUT_p;
7
8          SC_CTOR (testbench)
9          {
10             SC_THREAD (process);
11         }
12         ~testbench() { }
13
14         // Definition der Funktionalität vom "Prozess"-Thread
15         void process() {
16             while(1) {
17                 if( globalInputs[0] == LOGIC_0)
18                     {A0_p = SC_LOGIC_0;}
19                 . . .
20                 if( globalInputs[4] == LOGIC_1)
21                     {CIN_p = SC_LOGIC_1;}
22                 wait (10, SC_NS);
23                 if(SUM0_p == SC_LOGIC_0)
24                     {globalOutputs[0] = LOGIC_0;}
25                 . . .
26                 if(COUT_p == SC_LOGIC_1)
27                     {globalOutputs[2] = LOGIC_1;}
28                 GoOn = 0;
29                 while(GoOn == 0) { Sleep(0); }
30             }
31         }
32
33         int sc_main() {
34             sc_signal<sc_logic> A0_s, A1_s, B0_s, B1_s, CIN_s, SUM0_s, SUM1_s, COUT_s;
35
36             addtop add1 ("Addtop");
37             testbench test1("TestBench1");
38
39             add1 << A0_s << A1_s << B0_s << B1_s
40                 << CIN_s << SUM0_s << SUM1_s << COUT_s;
41             test1 << A0_s << A1_s << B0_s << B1_s
42                 << CIN_s << SUM0_s << SUM1_s << COUT_s;
43
44             sc_start(-1, SC_NS);
45             return(0);
46         }

```

---

Listing 6.3: Testbench im Interface-Teil

Im *SC\_MODULE(testbench)* findet man zunächst die Deklaration der Ein- und Ausgänge (Zeile 4, 5). Neben dem Konstruktor (Zeile 7) und dem Destruktor (Zeile 11) sieht man die Methode *process()*. In dieser Methode erfolgt die Anpassung und die Übergabe der Datentypen vom FIT zu SystemC und umgekehrt. Diese Aufgabe entspricht die eines Daten-Wrappers. Über das *GoOn*-Flag wird dem Scheduler mitgeteilt, dass die Methode fertig ist.

In der *sc\_main()*-Funktion erfolgt SystemC-typisch die Bildung von Signalen, die Instanziierung der Module *testbench* und der SystemC-Schaltung (Zeile 35, 36). Anschließend werden die Module miteinander verbunden (Zeilen 38...41). Der Befehl *sc\_start(-1, SC\_NS)* startet die SystemC-Simulation bis zur Terminierung durch den FIT-Simulator.

Die Fehlerinjektion geschieht im VHDL-Teil durch das FIT. In der SystemC-Komponente kann die Steuerung der Fehlerauslösung in dieser selbst implementiert sein (→5.4). Es besteht aber auch die Möglichkeit, über den VHDL-Simulator eine Fehlerliste einzulesen und über zusätzliche Ports diese Fehler in das SystemC-Modell zu injizieren.

## 6.5 Simulationsergebnisse

Die Kopplung der Hardwarebeschreibungen wurde mittels unterschiedlicher Implementierungen näher untersucht. Dabei kam es auch zu Betrachtungen verschiedener Abstraktionsebenen in Kombination mit diversen Eingangsstimuli. Die Tabelle 6.1 zeigt einige Simulationsergebnisse. Basis der Simulation ist ein 16-Bit-Addierer mit einem Simulationszeitraum von ca. einer Million Zyklen. Das Modell wurde als Verhaltensmodell (*behave*) und als Gattermodell (*gate*) erstellt.

Um einen Ergebnisvergleich zu erreichen, entstanden jeweils ein SystemC-Modell (*sc*) und eine Mixed-Modell (*mix*). Die Aufteilung der Aufgaben beim Mixed-Modell ist so, dass die Systembeschreibung auch als SystemC-Modell realisiert ist. Die Testbench-Pflichten wie Ein-/Ausgaben etc. wurden dagegen vom VHDL-Simulator übernommen. Da es sich bei allen Simulationen um ereignisgesteuerte Ausführungen handelt (→3.1.5.1), hat die Formierung der Eingangsmuster einen großen Einfluss auf den Simulationsablauf. Um hierzu eine nähere Aussage zu treffen, wurden zwei Randbedingungen mitbetrachtet. Die Musterfolge M1 entstand durch einen Zähler. Die Musterfolge M2 entstand durch einen Zufallszahlengenerator und besitzt eine sehr unregelmäßige Struktur, die den Simulator zu viel Aktivität verleitet. Die Musterfolge M3 zeigt sehr wenig Aktivität, in dieser ändert sich zwischen den aufeinander folgenden Mustern nur eine Position.

Modell	M1 (Zähler)	M2 (Zufall)	M3 (Einzelbit)
a16_sc_behave	5,2 s	7,2 s	5,2 s
a16_mix_behave	4,3 s	4,9 s	4,2 s
a16_sc_gate	5,9 s	16,7 s	5,3 s
a16_mix_gate	6,0 s	16,4 s	5,5 s

Tabelle 6.1: Simulationen der Mixed-Language-Co-Simulation

Es lässt sich erkennen, dass viel Aktivität, bedingt durch stark wechselnde Eingangsmuster M2, erwartungsgemäß zu einer höheren Simulationszeit führen. Besonders deutlich wird dies beim Gattermodell. Es zeigt sich auch, dass das Gattermodell bei geringer Aktivität nur unwesentlich mehr Zeit braucht, als das Verhaltensmodell. Die wichtigste Aussage steckt im Vergleich von der Mixed-Simulation zur puren SystemC-Simulation. Hier schneidet die Mixed-Simulation prinzipiell nicht schlechter ab als die SystemC-Modellierung. Für das Verhaltensmodell ergeben sich sogar leichte Vorteile.

In der Abbildung 6.3 sind die Ergebnisse aus der Tabelle 6.1 im Vergleich zu anderen Co-Simulationen in einem Diagramm dargestellt. Die vorgestellten Benchmark-Modelle sind recht unterschiedlich, dennoch lässt der Vergleich Schlussfolgerungen bezüglich der Effizienz der Mixed-Language-Co-Simulation zu. Die Modelle *MLCS-B* und *MLCS-G* sind die hier vorgestellten Mixed-Language-Co-Simulations-Beispiele. Das Modell *GPS* stammt aus [202] und steht für das Modell eines GPS-Receiver. Bezeichnend für die beiden GPS-Modelle ist, dass eine Teilschaltung einmal als SystemC-Register-Transfer-Modell und zum anderen als VHDL-Register-Transfer-Modell konstruiert ist.

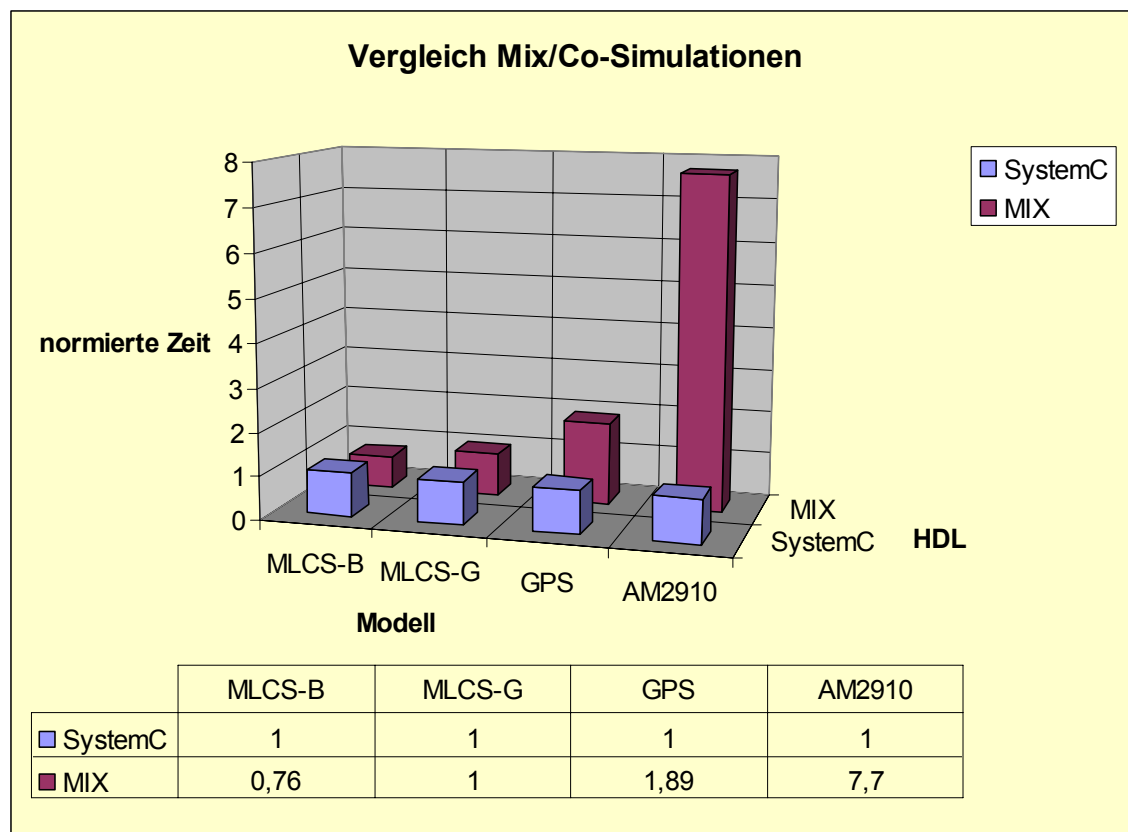


Abbildung 6.3: Vergleich verschiedener Mixed-Language/Co-Simulationen

Das Modell AM2910 wurde aus [38] entnommen. Bei dieser Schaltung handelt es sich um einen Mikroprogramm-Adress-Sequencer. Die Simulationszeiten wurden für einen besseren Vergleich auf die Simulationszeit des reinen SystemC-Modells normiert.

Es lässt sich gut erkennen, dass die hier vorgestellte Art der Co-Simulation verschiedener Hardwarebeschreibungen am besten abschneidet. Auch ohne die Zuhilfenahme eines Simulators mit einem Mixed-Language-Single-Kernel ergeben sich zwischen SystemC- und Mixed-Simulation keine Nachteile.

Statt der POSIX-Threads wurden auch Implementierungen mit Windows-Threads [218] erstellt und simuliert. Für diese Ausführungen ergaben sich im Vergleich zu den POSIX-Threads nahezu identische Ausführungszeiten. Aus Portabilitätsgründen wurde deshalb die POSIX-Variante bevorzugt.

Die hier vorgestellte Kopplung eines SystemC-Projektes mit einer Anwendung lässt sich auch für andere Applikationen nutzen. Hierzu müssen im Wesentlichen die Callback-Funktionen zur Kommunikation und Steuerung zwischen den beiden Anwendungen in der Interface-Beschreibung implementiert sein. So lässt sich zum Beispiel ein SystemC-Simulationsmodell mit einer Graphical-User-Interface (GUI) ausstatten.

## 7 Simulationsbeschleunigung durch Parallelität

Im Kapitel 5 wurde gezeigt, dass die simulierte Fehlerinjektion mit SystemC effektiv durchführbar ist. Dabei wird die klassische Fehlerinjektion (*fault injection*), welche eine Form der Simulation mit Fehlern darstellt, dazu benutzt, um das Verhalten eines Systems beim Auftreten eines Fehlers während des Betriebs näher zu untersuchen (→5.2). Typische Fehler in diesen Experimenten sind solche, die eine transiente Natur haben. Das Auftreten permanenter Fehler während der Laufzeit ist weitaus seltener. Eine vollständige Erfassung aller möglichen Fehlervarianten erweist sich schon für etwas größere Systeme als schwierig, da der Ereignisraum (→5.1.3) unter Berücksichtigung von Mehrfachfehlern, unterschiedlichen Fehlermodellen, usw. exponential wächst. Neben der Fehlerinjektion besitzt die Simulation von Fehlern für die Validierung von Testmustern für den Fertigungstest hohe Bedeutung. Dieses Gebiet wird üblicherweise als die Fehlersimulation (*fault simulation*) bezeichnet (→3.2). Im und nach dem Fertigungstest sind die permanenten Fehler, entstanden durch Defekte während des Produktionsprozesses, von Interesse. Bei der Simulation sind diese permanenten Fehler auch im Fokus der Ausführung. Eine einfache simulierte Fehlerinjektion wäre auch in der Lage eine Fehlersimulation durchzuführen. In der Realität muss eine Fehlersimulation aber auch größere Systeme in endlicher Zeit und für alle Fehler bewältigen. Ohne Simulationsbeschleunigung ist dieses Problem schwer zu lösen. Mit einer schnelleren Simulation von Fehlern wären nicht nur mehr Fehlerinjektionsexperimente in gleicher Zeit möglich oder die Ausführung von Experimenten in kürzerer Zeit. Auch der Einsatz der Fehlerinjektionstechniken (→5.4) zur Validierung von Testmustern oder dem Anlegen eines Fehlerlexikons (*fault dictionary*) wären Anwendungen [23].

Der Wunsch den Programmablauf auf einem Rechner zu beschleunigen, ist ein primärer Eckpunkt seit dem Beginn des Computerzeitalters. Zum einen hilft ein schnellerer Rechengvorgang, bedingt durch die kürzere Ausführung, Zeit und damit Kosten zu sparen. Zum anderen werden komplexe Probleme erst durch schnelle Abläufe lösbar. Außerdem spielt aus wirtschaftlichen Gesichtspunkten auch die schnelle Realisierung eines Projektes, also ein kurzes *time to market*, meistens eine große Rolle. Für die Simulation einer Schaltung mit der Injektion von Fehlern ist eine beschleunigte Ausführung von besonderer Bedeutung. Steigt doch mit der Anzahl der Fehlerinjektionen der Rechenaufwand proportional (→3.2). Oft wird erst mit einer beschleunigten Programmausführung aus einem simulierten Experiment mit

Fehlerinjektion (→5.2.1.3) der Übergang zu einer effizienten Fehlersimulation (→3.2) möglich.

Eine frühe Form der Beschleunigung ist die Parallelisierung von Rechengvorgängen (→3.2.2). Der Begriff Parallelisierung ist hierbei sehr allgemein. Zuweilen wird dieser Begriff auch für quasi-parallele Vorgänge benutzt, die nach außen parallel erscheinen, aber sequenziell abgearbeitet werden. In diesem Fall ist die Bezeichnung der konkurrenten (nebenläufigen) Ausführung treffender. Parallelität ist dadurch gekennzeichnet, dass sich der Ablauf von Prozessen mindestens teilweise überlappt. Um die hier vorgestellten Varianten der parallelen Anwendungen besser zu verstehen, werden zunächst einige Begriffe der Parallelverarbeitung in diesem Zusammenhang erläutert.

Die wichtigste Voraussetzung für die parallele Ausführung ist die Zerlegung einer Aufgabe in Teilaufgaben. Diese Teilaufgaben bilden Berechnungsströme, die auf physikalische Berechnungseinheiten abgebildet werden [70]. Dieser Vorgang wird auch als *Mapping* bezeichnet. Die Berechnungsströme können einerseits aus den zu verarbeitenden Daten bestehen oder aus den Instruktionen des Programms gebildet werden, also durch die Zerlegung des Programmflusses.

Wesentliche Aufgaben, die es neben der Ausführung von Abschnitten zu lösen gilt, sind die Kommunikation und die Synchronisation. Bei der Ausführung von *Abschnitten*, welche Teilfolgen von Aktionen bilden, gibt es solche, die kritisch zueinander sind [71]. Solche *kritischen Abschnitte* dürfen nicht parallel zueinander ablaufen und müssen durch Schutzmechanismen gesichert werden. Hierbei entsteht eine Zwangssequenzialisierung. *Kommunikation* bezeichnet jegliche Art des Datenaustausches zwischen den Prozessen. Bei der Haltung der Daten haben sich zwei Konzepte durchgesetzt. Zum einen gibt es das Modell mit einem gemeinsamen Adressraum und zum anderen das Modell mit einem verteilten Adressraum. Beim Ansatz mit dem gemeinsamen Speicher kann die Datenübertragung zwischen den Prozessen auch über gemeinsame Variablen etc. erfolgen. Bei getrennten Speicherbereichen kommunizieren die Prozesse mittels Nachrichten (messages). Damit die Prozesse, insbesondere bei der Abarbeitung kritischer Abschnitte, korrekt Daten austauschen, ist eine geeignete *Synchronisation* erforderlich. Die Synchronisation ist eine besondere Form der Kommunikation bei der keine Benutzerdaten, sondern nur Informationen zum System ausgetauscht werden. Solche Informationen sind z. B. Barrieren, bei der alle beteiligten Prozesse warten müssen, bis alle den Synchronisationspunkt erreicht haben.

Nachdem die Entscheidung für das gewählte Programmiermodell fest steht, gilt es drei Hauptpunkte zur Entwicklung des parallelen Programms auszuführen [72]:

- *Zerlegung*: Die Anwendung muss in eine Gruppe von parallelen Prozessen und Daten aufgeteilt werden.
- *Zuordnung*: Die Prozesse und Daten werden auf die Knoten verteilt.

- 
- *Feinabstimmung*: Die Arbeitsweise der Anwendung wird geändert, um die Leistung zu steigern. Hierzu zählen z. B. das Ausgleichen der Rechenlast, das Verringern der Kommunikation und das Vermeiden sequenzieller Engpässe.

Eine einheitliche Unterteilung parallel ablaufender Programme ist in der Literatur nicht zu finden. Es existiert allerdings eine 4-Ebenen-Einteilung, mit der sich fast alle parallelen Anwendungsfälle bezüglich ihres Parallelisierungsgrades klassifizieren lassen [66]. Die Abbildung 7.1 zeigt das entsprechende Schichtenmodell.

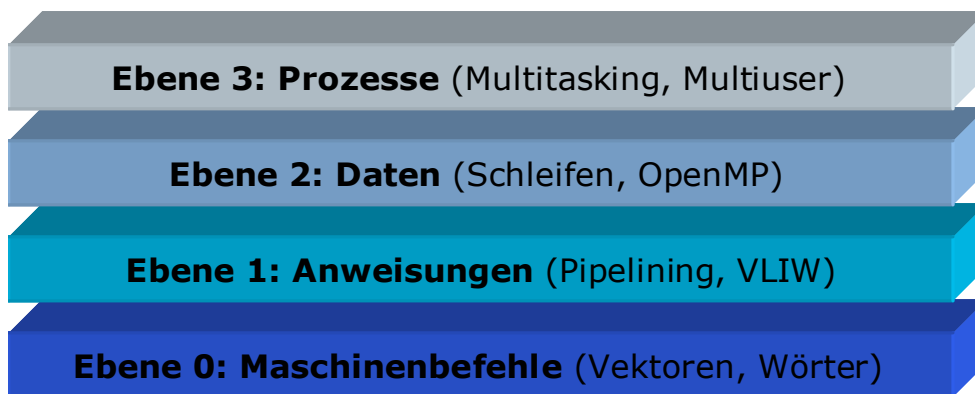


Abbildung 7.1: Einfaches Schichtenmodell zum Parallelisierungsgrad

Die **Ebene 3** ist die höchste Schicht in der Abbildung 7.1. In dieser Ebene werden unabhängige Prozesse parallel abgearbeitet. Die Prozesse können dabei von einem Anwender oder verschiedenen Nutzern (Multi-user) sein. Typisch für einen Rechner ist hierbei der Multitasking-Betrieb oder bei einer feineren Auflösung das Multithreading (→6.3). Prozesse können auch auf unterschiedlichen CPUs in einem Netz ausgeführt werden, wie z. B. in einem PC-Cluster. Für schwergewichtige Prozesse ist es typisch, dass jeder über einen unabhängigen Prozessadressraum verfügt. Die Parallelisierung ist in dieser Ebene häufig gut skalierbar.

Die **Ebene 2** bezeichnet die Möglichkeit der Ausnutzung von Datenparallelität bzw. die Parallelisierung anhand von Datenstrukturen. Zum Beispiel werden hierbei Datenfelder, welche in Form von Vektoren oder Matrizen vorliegen können, in Schleifen berechnet. Finden sich in den Datenfeldern unabhängige Berechnungsschritte und im Rechnersystem mehrere Prozessorkerne wieder, so lässt sich dieses Verfahren gut anwenden und skalieren. Eine oft verwendete Umgebung dieser Ebene ist das OpenMP [67,65], welches per *Pragmas* im Programmcode eine manuelle Parallelisierung erlaubt. Moderne Compiler bieten eine automatisierte Parallelisierung mittels OpenMP, ohne dass der Anwender hierzu über spezielle Kenntnisse verfügen muss [69].

In der **Ebene 1** liegt die Parallelität mit den einzelnen Instruktionen vor. Im Allgemeinen wird diese Form von parallelisierenden Kompilern gut unterstützt. Dies findet man z. B. bei VLIW-Prozessoren vor. Diese Prozessoren enthalten mehrere unabhängige Funktionseinheiten wie ALUs oder FPUs, welche parallel unabhängige Anweisungen ausführen. Eine andere und sehr häufig in dieser Ebene anzutreffende Variante ist das Pipelining. Hierbei wird ein Befehl in Teilanweisungen zerlegt, die dann von zugeordneten HW-Einheiten, den sogenannten Pipelinestufen, ausgeführt werden. Die Skalierung ist bei der Ebene 1 schlechter als in den Ebenen 2 und 3. Für den Programmierer bzw. Anwender ergeben sich wenig Einflussmöglichkeiten, da die Parallelisierung durch die gewählte CPU und dem Compiler vorgegeben ist. Bei genauerer Kenntnis der Arbeitsweise der CPU lässt sich u. U. Programmcode schreiben, der die Effizienz der Parallelisierung verbessert oder verschlechtert.

In der **Ebene 0** wird die Parallelität der Wörter eines Maschinenbefehls genutzt. Es besteht eine Parallelität auf der Bitebene. Typisch hierfür ist in wissenschaftlichen Anwendungen die Verwendung von Vektorprozessoren. Die Skalierbarkeit der Parallelisierung hängt direkt von der Bitbreite des Prozessors ab und ist somit stark eingeschränkt. Heutige Prozessoren nutzen eine Breite von 32 oder 64 Bits. Eine weitere Erhöhung der Bitbreite und somit der Parallelität ist bei General-Purpose-Prozessoren in nächster Zeit nicht zu erwarten. Die Breite von 64 Bits ist selbst für Gleitkommazahlen bei den meisten Anwendungen ausreichend.

Eine typische Eigenschaft für die Kategorisierung nach Ebenen ist die *Granularität*. Je nach Ebene ist sie unterschiedlich. Bestehen die Module, die parallel ablaufen, aus wenigen Instruktionen, so ist die Granularität sehr fein. Im Gegenzug bezeichnet man die Granularität bei vielen Instruktionen als grob. Die Granularität wird auch als durchschnittliche Größe der Teilaufgaben angesehen. Für das Ebenenmodell bedeutet dies, dass die Ebene 0 eine sehr feine und die Ebene 3 eine sehr grobe Granularität besitzt.

Bezüglich ihrer Organisation der globalen Kontrolle und den sich ergebenden Daten- und Kontrollflüssen werden Parallelrechner nach Flynn [73] theoretisch in vier Klassen unterschieden:

**SISD** (*Single Instruction Single Data*): Dieser Rechner hat einen Prozessor und einen Daten- und einen Programmspeicher. Dieses Modell entspricht dem klassischen *von-Neumann*-Rechnermodell.

**MISD** (*Multiple Instruction Single Data*): Dieses Modell besitzt mehrere Verarbeitungseinheiten mit jeweils einem eigenen Programmspeicher. Es gibt allerdings nur einen gemeinsamen Zugriff auf den Datenspeicher. Ein Datum wird von mehreren Prozessoren, mit u. U. unterschiedlichen Instruktionen, gleichzeitig berechnet und wieder abgespeichert. Die entstehenden Ergebnisse sind im Normalfall identisch. Dieses Modell hat für Parallelrechner keine praktische Bedeutung [70]. Das liegt sicher daran,



---

dass sich keine Steigerung der Ausführungszeit ergibt. Für den Einsatz in verlässlichen und fehlertoleranten Systemen kann die Verwendung dieses Modells durchaus sinnvoll sein (→5.1.1). Die in dieser Variante vorliegende Redundanz in der Berechnung lässt sich z. B. zur Fehlerkorrektur nutzen [151].

**SIMD** (*Single Instruction Multiple Data*): Dieses Modell besteht aus mehreren Verarbeitungseinheiten, bei der jede Einheit einen eigenen Zugriff auf den Datenspeicher besitzt. Allerdings gibt es nur einen Programmspeicher, so dass jeder Prozessor die gleiche Instruktion erhält, aber separate Daten verarbeitet. Die Synchronität ergibt sich hierbei automatisch.

**MIMD** (*Multiple Instruction Multiple Data*): Dieses Modell besteht aus mehreren Prozessoren, die jeweils einen eigenen Zugriff auf einen Datenspeicher haben. Jeder Prozessor besitzt zudem einen eigenen lokalen Programmspeicher. Die Verarbeitungseinheiten können durchaus asynchron zueinander arbeiten. Dies erfordert zusätzliche Mechanismen, um die Synchronität zu erreichen.

Da neben der Programmausführung in den CPUs auch der Datenaustausch zur Leistungsfähigkeit des Gesamtsystems beiträgt, bekommt der Kommunikationsaufwand u. U. eine große Bedeutung. Dabei trägt für Multiprozessorsysteme und Multicomputersysteme das Verbindungsnetzwerk einen beträchtlichen Anteil zur Leistung bei. Das Netzwerk wird wesentlich durch die beiden Gestaltungskriterien Topologie und Routingtechnik bestimmt [66]. Die *Topologie* beschreibt die Verschaltung der Prozessoren und Speicherkomponenten. Die *Switching- und Routingtechnik* bezeichnet die Strategie der Nachrichtenübermittlung zwischen den Prozessoren bzw. Speichern.

Ein weiteres Merkmal des parallelen Rechnens ist die Lastverteilung (*load balancing*) auf die einzelnen Prozessoren. Bei gleichmäßiger Lastverteilung sind alle Prozessoren stets beschäftigt. Durch Wartevorgänge, wie z. B. durch eine nicht optimale Verteilungsstrategie oder durch Synchronisationsprozesse, kann dieser ideale Zustand häufig nicht erreicht werden.

Der verbleibende Teil des Kapitels ist so aufgebaut, dass zunächst einige wichtige Kenngrößen der Parallelverarbeitung vorgestellt werden. Anschließend folgen verschiedene Implementierungen für die Simulationsbeschleunigung für ein Einprozessorsystem und danach eine Umsetzung des verteilten Rechnens auf einem PC-Cluster.

## 7.1 Wichtige Kenngrößen des parallelen/verteilten Rechnens

Zur Bewertung der Leistungsfähigkeit paralleler Anwendungen existieren diverse Metriken. Ein Maß, welches zu den wichtigsten zählt, ist die Beschleunigung. Sie wird gemeinhin auch als Speed-up  $S_p$  bezeichnet. Sie ergibt sich aus der Ausführungszeit des Programms in serieller Form  $T_{ser}$  und aus der Ausführungszeit des parallelen Programms  $T_{par}$ . Allgemein lautet die Formel für den Speed-up:

$$S_p = \frac{T_{ser}}{T_{par}} \quad (7.1)$$

Für ein Problem fester Größe und mit homogener Struktur ergibt sich der Speed-up wie folgt:

$$S_p(P) = \frac{T_{par}(1)}{T_{par}(P)} \quad (7.2)$$

$P$  bezeichnet in der Formel 7.2 die Anzahl der Prozessoren. Im Idealfall wird ein Problem bei gleicher Problemgröße  $P$ -mal schneller ausgeführt. Einen derart guten Speed-up erreicht man in der Praxis fast nie. Das liegt zum einen daran, dass sich ein Problem nicht vollständig parallel ausführen lässt und zum anderen, dass zusätzlicher Overhead für die Kommunikation besteht. Dieser Overhead geht häufig zu Lasten der Problemberechnung, da die CPU Teile der Kommunikation übernimmt.

Zur einfachen Ermittlung des Speed-ups gilt es, noch einen weiteren Umstand zu berücksichtigen. Ein ursprüngliches, seriellles Programm mit der Laufzeit  $T_{ser}$  besitzt u. U. eine andere Laufzeit als ein paralleles Programm, dass nur einen Prozessor  $P$  nutzt und somit ebenfalls vollständig in der Zeit  $T_{par}(1)$  das Problem seriell löst.

$$T_{ser} \neq T_{par}(1) \quad (7.3)$$

Dieser Umstand entsteht dadurch, dass zur Implementierung der Parallelität in einer Anwendung zusätzliche Modifikationen nötig sind und sich hieraus ein Mehraufwand an Ausführungszeit ergibt. Bei der Ermittlung des Speed-ups ist es also notwendig zu berücksichtigen, gegen welche serielle Größe die parallele Beschleunigung ermittelt wird. Neben der Betrachtung des Speed-ups für eine feste Problemgröße kann man die Beschleunigung auch für eine feste Laufzeit untersuchen [66]. In diesem Fall lässt sich durch eine höhere Prozessoranzahl eine bessere Genauigkeit erreichen. Für die Applikationen in dieser Arbeit besteht vornehmlich die Aufgabe, ein Problem fester Größe zu lösen.

Eine weitere wichtige Größe einer parallelen Ausführung ist das Maß der Effizienz  $E_{par}$ . Sie ist für eine feste Problemgröße und einem homogenen System als Beschleunigung pro Knoten definiert:

$$E_{par}(P) = \frac{S_{par}(P)}{P} \quad (7.4)$$

In den meisten Anwendungen ist die Effizienz kleiner 1. Ist die Effizienz  $E_{par}(P) = 1$  und  $P \neq 1$ , so skaliert das System optimal proportional zur Problemgröße. Systeme die eine Effizienz von größer eins aufweisen, werden als superskalar oder hyperskalar bezeichnet.

Ein ähnliches Gütemaß wie die Effizienz, ist die Größe der Kosten  $C_{par}$  eines parallelen Programms:

$$C_{par}(P) = T_{par}(P) \cdot P \quad (7.5)$$

Es ist ein Maß für die durchgeführte Arbeit aller Prozessoren. Ein paralleles Programm ist kostenoptimal, wenn gilt:

$$C_{par}(P) = T_{ser}(P) = T_{ser} \cdot P \quad (7.6)$$

$T_{ser}(P)$  mit  $P = 1$  bezeichnet die Laufzeit einer optimalen sequenziellen Lösung des Problems.

Im Allgemeinen ist es nicht möglich, ein Problem mit beliebiger Prozessoranzahl und einer hohen parallelen und numerischen Effizienz bei vernachlässigbarem Kommunikationsaufwand zu lösen. Mit zunehmender Prozessoranzahl nehmen gewöhnlich die Skalierbarkeit und die Effizienz ab. Dieser Umstand wurde von Amdahl [75] recht früh untersucht. Die Betrachtungen führten zur Definition des *Amdahl'schen Gesetzes* [70]:

$$S_p(P) = \frac{1}{f + \frac{1-f}{P}} \leq \frac{1}{f} \quad (7.7)$$

Die Größe  $f$  ( $0 \leq f \leq 1$ ) bezeichnet einen Bruchteil der parallelen Implementierung, der sequenziell ausgeführt werden muss. Nicht berücksichtigt wurde in der Formel 7.4 der Aufwand für die Kommunikation, dieser verringert den Speed-up zusätzlich. Aus der Formel 7.7 ergibt sich ein oberer Grenzwert für den Speed-up von:

$$\lim_{P \rightarrow \infty} S_p(P) = \lim_{P \rightarrow \infty} \frac{1}{f + \frac{1-f}{P}} = \frac{1}{f} \quad (7.8)$$

Aus der Formel 7.8 lässt sich erkennen, dass eine hohe Anzahl von Prozessoren keine Beschleunigung bewirkt. Amdahl wollte damit zeigen, dass sich ein Problem nur bedingt sinnvoll parallel ausführen lässt und der Speed-up einen asymptotischen Verlauf hat. Bei einem sequenziellen Anteil  $f$  von 20 % wäre eine Prozessoranzahl von maximal fünf sinnvoll. Amdahl schlussfolgerte, dass große parallele Systeme viele Probleme nicht schneller lösen können als kleinere Systeme.

In der vorhergehenden Betrachtung wurde von Amdahl davon ausgegangen, dass die Problemgröße von der Anzahl der verwendeten Prozessoren unabhängig ist. In der Praxis jedoch werden bei der Verwendung von leistungsfähigeren Systemen auch die Ansprüche erhöht. Man berechnet zum Beispiel ein Problem mit einer höheren Genauigkeit, was eine Verringerung des sequenziellen Anteils  $f$  bewirkt. Diesen Ansatz haben Gustafson et al aufgegriffen und die Problemgröße berücksichtigt [68]. So wurde angenommen, dass der parallele Anteil mit der Problemgröße skaliert. Somit lautet nach Vereinfachungen das Gustafson-Gesetz:

$$S_p(P) = P + (1 - P) \cdot t_{ser} \quad (7.9)$$

Die Größe  $t_{ser}$  bezeichnet in der Formel 7.9 die im seriellen Teil verbrachte Zeit. Für eine hohe Parallelisierbarkeit läuft der Grenzwert des Speed-ups idealerweise gegen die Prozessoranzahl.

$$\lim_{P \rightarrow \infty} S_p(P) = P \quad (7.10)$$

Die Formel 7.10 zeigt augenscheinlich, im Gegensatz zur Formel 7.8, dass sich Anwendungen gut parallelisieren lassen. Laut [64] lassen sich beide Ansätze ineinander überführen. So stellen die beiden Formeln 7.10 und 7.8 Grenzfälle dar, wobei 7.8 als untere und 7.10 als obere Schranke zu betrachten ist. Wie gut sich eine Anwendung wirklich parallelisieren lässt, hängt von der Problemstellung, den vorhandenen Mitteln und der Umsetzung ab.

## 7.2 Beschleunigung durch paralleles Rechnen auf einem Einzelprozessorsystem

Im vorhergehenden Abschnitt wurden einige Begriffe zum parallelen Rechnen vorgestellt. Im Folgenden werden einige Implementierungen aufgeführt, die verschiedene Wege gehen, um eine Beschleunigung der Simulation zu erreichen. Den hierfür vorgestellten Lösungen ist zu eigen, dass sie die Datenparallelität auf einer CPU bzw. in einem Computersystem nutzen. Diese Arbeiten werden deshalb dem *SIMD*-Rechnermodell zugeordnet. Dabei unterscheidet sich die Granularität je nach Implementierung. Des Weiteren muss keine Berücksichtigung der Synchronität erfolgen, da diese durch die Berechnungsmodelle vorgegeben ist. Zusätzliche HW-Kosten entstehen bei den vorgestellten Verfahren nicht, da nur ungenutzte Ressourcen des Rechensystems verwendet werden und/oder der Programmablauf eine andere Organisation erfährt. Bisher sind keine vergleichenden Untersuchungen auf dieser Ebene in SystemC bekannt.

Der verbleibende Abschnitt ist so aufgebaut, dass insgesamt drei Ansätze mit jeweils zwei Lösungen zur Vorstellung kommen. Diverse durchgeführte Beispielsimulationen lassen Rückschlüsse auf die Leistungsfähigkeit und die Grenzen der Parallelisierbarkeit der einzelnen Methoden zu.

### 7.2.1 Bit-parallele Simulation

Bei einer sequenziellen Simulation von zweiwertiger (bivalenter) Logik wird normalerweise nur ein Bit mit einer Instruktion berechnet ( $\rightarrow$ 3.2.1). Eine der einfachsten Formen, eine Parallelisierung zu erreichen, ist die Simulation von mehreren Bits in einem Maschinenwort. Dieses Verfahren bietet sich für die zweiwertige Logik geradezu an, da die gültigen logischen Operationen wie AND, OR, usw. für eine Bitstelle keinen ungewollten Einfluss auf die Nachbarbits haben. Außerdem stehen die logischen Operationen bei *General-Purpose*-Prozessoren als direkte Maschinenbefehle bereit, was die Umsetzung erleichtert. Der Speed-up  $S_p$  ist hierbei im Idealfall so hoch wie die Anzahl der verwendeten Bits  $A_b$ :

$$S_p = A_b \cdot T_{ser} \quad (7.11)$$

Eine obere Schranke hinsichtlich der Skalierbarkeit entsteht durch die Breite eines Maschinenwortes der CPU. Bezüglich der Parallelisierungsebene befindet sich die Bit-parallele Simulation auf der Ebene 0. Sie ist somit von der Granularität sehr fein.

Bezüglich der Entwurfsebene ist diese Technik der Gatter- und Schalterebene vorbehalten, da Logikdatentypen z. B. keine arithmetischen Operationen erlauben.

### 7.2.1.1 Musterparallele Simulation

Die musterparallele Simulation wurde bereits im Abschnitt 3.2.2.1 allgemein vorgestellt. Für die SystemC-Modelle wurde sie hier für kombinatorische Schaltungsmodelle umgesetzt, da für diese eine zeitliche Unabhängigkeit der Muster besteht. Mit dieser Eigenschaft erlauben sie eine gleichzeitige Ausführung. Bei sequenziellen Schaltungen führt die musterparallele Ausführung üblicherweise zu Fehlern. Dort haben die Schaltungszustände eine zeitlich sequenzielle Abhängigkeit. Wird nun parallel ausgeführt, so arbeitet das Nachbarbit u. U. mit einem Wert, der noch gar nicht bekannt sein kann.

Es lassen sich jedoch auch sequenzielle Schaltungen sinnvoll musterparallel berechnen. Dies ist der Fall, wenn es Gruppen von Eingangsmustern gibt, sogenannte unabhängige Sequenzen, die nichts miteinander zu tun haben. Für eine Mikroprozessorschaltung können dies z. B. verschiedene Programme sein, die simuliert werden müssen. In diesem Fall ist eine gleichzeitige Ausführung möglich. Ein Scheduling von Programmen mit ähnlich langen Eingangssequenzen führt zu einer ausgeglichenen Lastverteilung (balanced computation) und damit zum höchsten Speed-up. Ein Vorteil der musterparallelen Simulation besteht darin, dass es bezüglich der Parallelisierung keiner Modifikation des Modells bedarf. Einzig beim Zusammenstellen der Testbench müssen die modifizierten Eingangs- und Ausgangsstimuli berücksichtigt werden.

### 7.2.1.2 Fehlerparallele Simulation

Neben der parallelen Berechnung von Eingangsmustern lassen sich auf der Bitebene auch Fehlerinjektionen parallel ausführen, da sie voneinander unabhängig sind. Für ein Eingangsmuster werden mehrere Fehlerzustände simultan berechnet. Ein zusätzlicher Aufwand bei dieser Methode besteht in der parallelen Injektion der Fehler. Während bei der musterparallelen Abarbeitung nur die Eingangsmuster und eventuell auch die Ausgaben sortiert werden mussten, ist hier ein erweitertes Scheduling der Fehlerinjektion und die Duplizierung eines Eingangswertes auf alle Bitstellen des Eingabe-Bytes nötig.

Die Umsetzung der Fehlerinjektion kann auf verschiedene Weise erfolgen. Die universellste Methode ist die mit bedingten Anweisungen. Für das Haftfehlermodell ist die Verwendung logischer Operationen zumeist der effizienteste Weg. Das Listing 7.1 zeigt die Implementierung für einen Saboteur mittels logischer Operationen.

Ein stuck-at-1-Fehler wird durch eine OR-Operation aus dem Fehlerselektionswort *sal* (Zeile 6) und dem Eingangswert *in* (Zeile 4) injiziert (Zeile 18). Einen stuck-at-0-

Fehler verwirklicht man durch eine AND-Operation (Zeile 17) aus dem Eingangswert *in* (Zeile 4) und dem Einer-Komplement des Fehlerselektionswortes *sa0* (Zeile 5). Eine '1' an der entsprechenden Bitstelle des Fehlerselektionswortes injiziert den gewünschten Fehler.

---

```
2  SC_MODULE(sasab) {
    sc_out<unsigned int> out;
4   sc_in<unsigned int> in;
    sc_in<int> sa0;           //zur Injektion von sa0
6   sc_in<int> sa1;           //zur Injektion von sa1

8   unsigned int var_in, var_out;
    int var_sa0, var_sa1;
10  . . .
}

12 void sa0lsab::inject() {
    var_in = in.read();
14  var_sa0 = sa0.read();
    var_sa1 = sa1.read();
16
    var_out = (var_in & (~var_sa0)); //sa0
18  var_out = (var_out | var_sa1);   //sa1
    out.write(var_out);
20 }
. . .
```

---

Listing 7.1: Schneller fehlerparalleler Saboteur in SystemC

Neben den logischen Operationen zur Fehlerinjektion sind noch weitere Verfahren möglich. Beispielsweise kann dies per Abfrage mit *switch-case*-Kommandos, wie im Listing 5.1 zu sehen, ausgeführt werden. Da die Anzahl der nötigen Operationen linear mit der Parallelisierung steigt, ist dieses Vorgehen aus Zeitgründen nicht empfehlenswert.

Eine andere beliebte Realisierung ist die tabellengesteuerte Methode, da sie schnell in ihrer Ausführungszeit ist. Hier wird durch die Übergabe von Parametern aus einem *const*-Array ein entsprechender Tabelleneintrag geliefert. Jedoch wächst mit zunehmender Bitbreite, die bei der Parallelisierung erwünscht ist, und der Verwendung aller Möglichkeiten der Speicherbedarf exponential. Bei einem typischen 32-Bit-Maschinenwort sind dies schon  $2^{32}$ -Möglichkeiten. Da der Platzbedarf sowohl für den Eingangswert als auch für den Fehlerinjektionswert steigt, besitzt eine vollständige Tabelle  $2^{33}$  Einträge. Also ist dieser Weg aus Speicherplatzgründen noch weniger geeignet.

Einige Fehlersimulatoren arbeiten nur deshalb so effektiv, weil sie das Einzelfehlermodell (Single Fault Propagation Model) benutzen [21]. Für die Validierung von Testmustern für den Produktionstest mag dies ausreichend sein. Für die Simulation bei der Entwurfsvalidierung mit beispielsweise dem Auftreten mehrerer

gleichzeitiger Fehler ist dies untauglich. Das hier vorgestellte fehlerparallele Verfahren in SystemC erlaubt aber beliebig viele Fehlerinjektionen für einen Simulationsschritt und ist damit vielseitiger. Ein anderer Vorteil der fehlerparallelen Methode liegt im Vergleich zu musterparallelen Varianten darin, dass sich auch sequenzielle Schaltungen simulieren lassen. Die zeitlichen Abhängigkeiten der Schaltungszustände bleiben durch die sequenzielle Abarbeitung der Eingangsmuster erhalten.

### 7.2.1.3 Bit-parallele Beispielsimulationen

Die Tabelle 7.1 zeigt einige Simulationsergebnisse für kombinatorische Schaltungen in SystemC. Die Länge der Simulation wurde in Hinsicht auf zeitlich messbare Ergebnisse geformt, ansonsten wäre für das Modell *c17* aufgrund ihrer kleinen Schaltungsgröße kein sinnvolles Messergebnis entstanden. Die Schaltung *MSAB* ist größer als die Schaltung *c17*. Von Interesse ist in der Tabelle 7.1 der erreichte Speed-up  $S_p$ .

Design	C1	C32i	C32f
	Speed-up	Speed-up	Speed-up
c17	1	23,5	21,6
MSAB	1	24,6	26,0

Tabelle 7.1: Simulationsergebnisse für Bit-parallele Beispiele

Das Modell *C1* benutzt keine Parallelität, sondern ist sequenziell und hat die Ausführungszeit  $T_{ser}$  (→Formel 7.3). Die Variante *C32i* verwendet 32 verschiedene Eingangsmuster in einem Simulationsschritt und *C32f* rechnet simultan mit 32 Fehlern. Die Effizienz liegt unterhalb des theoretischen Maximums von 32. Zum einen ist dies bedingt durch den zusätzlichen Verwaltungsaufwand und zum anderen durch die Wahrscheinlichkeit von viel mehr auftretenden Ereignissen. Dies führt zu mehr Einträgen in der Ereigniswarteschlange und somit zu mehr Tätigkeit für den Simulationskernel (→3.1.5).



## 7.2.2 Simulationen mit vierwertiger Logik

Für eine funktionale Logiksimulation oder eine einfache Fehlersimulation mit Haftfehlern mag die zweiwertige Simulation genügen. Für einen brauchbaren Fehlersimulator ist der Gebrauch eines 3- oder 4-wertigen Logik-Datentyps unumgänglich. Wird beispielsweise ein Fehlersimulator in der Automatischen Test Pattern Generierung genutzt, so finden dort unbekannte 'X'-Werte Verwendung, um nicht determinierte Eingänge, bezogen auf einen Zielfehler, zu charakterisieren. Diese Information wird verwendet, um Test-Pattern-Sätze zu komprimieren. Andererseits ist auch bei der Simulation von Fehlern für die Abschätzung der Fehlertoleranz eines Systems das Rechnen mit unbekannten Werten nötig, da hier Signale kollidieren können. Solche Kollisionen lassen sich in den meisten Fällen nicht mit einer bivalenten Logik darstellen. Bezüglich der Entwurfsebene gilt das Gleiche wie für die Bit-parallelen Verfahren, sie arbeiten vorzugsweise auf der Gatter- oder Schalterebene.

### 7.2.2.1 Parallele Simulation mit SystemC-Logikvektoren

SystemC bietet als vierwertigen Datentyp *sc\_logic* bzw. *sc\_signal\_resolved* an. Um beispielsweise Busstrukturen zu simulieren, gibt es die Vektortypen *sc\_lv<>* und *sc\_rv<>*. Diese Datentypen bieten sich allerdings auch zum Parallelisieren der Simulation an. Für die Implementierung dieses Verfahrens müssen nur Signale, Variablen und Ports auf den Vektortyp umgestellt werden. Zum Beispiel wird bei der Deklaration eines Signals aus:

```
sc_in<sc_logic> IN1; ein sc_in<sc_lv<8> > IN1;.
```

Diese Änderungen lassen sich z. B. mittels (Perl-)Skripte [184] gut automatisieren. Die grundlegenden Methoden müssen nicht angepasst werden, da die logischen Operationen bei *sc\_logic* und *sc\_rv<>* bzw. bei *sc\_signal\_resolved* und *sc\_lv<>* gleich arbeiten, bzw. die gleichen Operatoren verwendet werden. Einzig die Datenanpassung für die Vektorbildung muss in der Testbench-Gestaltung erfolgen.

### 7.2.2.2 Parallele Simulation mit selbstdefiniertem Logikvektor

Ein anderes Verfahren zur parallelen Berechnung mit vierwertiger Logik ist an die Wirkungsweise angelehnt, wie es sie in anderen Fehlersimulatoren, wie z. B. dem Simulator PROOFS [22], Verwendung findet. Eine Umsetzung dieses Paradigmas, auch zum Zwecke einer Simulationsbeschleunigung, ist für SystemC bisher noch nicht bekannt. Die Kodierung eines Wertes erfolgt hierfür durch jeweils ein Bit an der gleichen Stelle in zwei Maschinenworten (*Z0,Z1*). Dabei lautet die Kodierung der Werte

wie folgt: '0' ist (1,0), '1' ist (0,1), 'X' ist (0,0) und 'Z' ist (1,1). Diese Form der Kodierung ist optimal, um mit wenigen logischen Operationen die Ziel-Operationen abzubilden. Um zum Beispiel eine Inverter-Funktion zu implementieren, werden nur die beiden Zuweisungen  $Z0 = A1$  und  $Z1 = A0$  verwendet.  $A0$  und  $A1$  sind die Eingabevektoren. Weitere Eingabevektoren werden symbolisch in alphabetischer Reihenfolge fortgeführt ( $B0, B1$ , etc.).

Die Tabelle 7.2 zeigt die Realisierung der logischen Ziel-Operationen mit Hilfe des selbstdefinierten Logikvektors. Eine Operation benutzt zur Eingabe die Vektoren  $A0, A1$  und  $B0, B1$ . Das Ergebnis wird in  $Z0$  und  $Z1$  abgelegt.

Operation	Z0	Z1
AND	$A0 \mid B0$	$A1 \& B1$
NAND	$A1 \& B1$	$A0 \mid B0$
OR	$A0 \& B0$	$A0 \mid B0$
NOR	$A1 \mid B1$	$A0 \& B0$
INV	$A1$	$A0$
XOR	$(A0 \& B0) \mid (A1 \mid B1)$	$(A0 \& B1) \mid (A1 \mid B0)$

Tabelle 7.2: Operationsauflösung des selbstdefinierten Logikvektors

Diese Form der Parallelisierung ließ sich in SystemC gut umsetzen und gebrauchen. Ein Eingabevektor-Paar selbst wurde als ein Array mit einer Feldgröße von zwei realisiert. Die symbolischen Darstellungen  $A0, A1$  wurden z. B. zweckmäßig zu `sc_signal<unsigned int> A0[2]` geformt. Wobei das  $A0$  dann zu  $A0[0]$  und  $A1$  zu  $A0[1]$  wurde. Das Listing 7.2 zeigt einen Teil der Umsetzung der 2-fach-NOR-Funktion mit den selbstdefinierten Vektoren.

---

```
2  SC_MODULE (nor2) {
    sc_out<unsigned int> z0[2];
4   sc_in<unsigned int> a0[2], a1[2];
    void doit();
6   SC_CTOR (nor2) {
        SC_METHOD (doit);
8       sensitive << a0[0]<< a0[1] << a1[0]<<a1[1];
        . . .
    }

10 void nor2::doit() {
    z0[1].write(a0[0].read() & a1[0].read());
12    z0[0].write(a0[1].read() | a1[1].read());
    }
```

---

Listing 7.2: NOR-Funktion mit selbstdefiniertem Vektor

In der Zeile 3 ist das Ausgabevektor-Paar definiert und in der Zeile 4 die Eingangsvektoren. Die Zeilen 11 und 12 sind für die Durchführung der NOR-Funktion relevant.

### 7.2.2.3 Vektor-parallele Beispielsimulationen

Die Tabelle 7.3 zeigt Simulationsergebnisse für einige kombinatorische Schaltungen. Der Simulationszeitraum wurde so gewählt, dass zeitlich messbare Ergebnisse entstehen. Die Schaltung *c17* ist die kleinste und *MSAB* ist die größte. Von Interesse ist in der Tabelle 7.3 der erreichte Speed-up  $S_p$ .

Design	sc_logic (1)	sc_lv<32>	sc_sd<32>
	Speed-up	Speed-up	Speed-up
c17	1	2,83	2,95
fsm6	1	7,84	9,08
decod	1	6,28	6,95
MSAB	1	2,27	3,81

Tabelle 7.3: Simulationsergebnisse für Vektor-parallele Beispiele

Die Variante *sc\_logic(1)* ist die einfache serielle Lösung mit einem Speed-up von eins. Die Methode *sc\_lv<32>* nutzt den Datenvektor von SystemC und *sc\_sd<32>* steht für die Lösung mit dem selbstdefinierten Vektor. Die Vektorlänge war jeweils 32 Bit. Es zeigt sich, dass der selbstdefinierte Datentyp unabhängig von der Schaltungsgröße die beste Performance bietet. Die Speed-up ist jedoch nicht konstant für die verschiedenen Modelle. Einen wesentlichen Einfluss auf die Performance hat auch die Datenein- und ausgabe. Je kleiner der Anteil an der Datenausgabe war, um so besser war der Speed-up. Nachteilig bleibt aber bei einer Lösung mit dem selbstdefinierten Vektor, dass dieser Typ nicht zu den C++/SystemC-Typen kompatibel ist. Treten neben dem selbstdefinierten Vektor noch andere Datentypen auf und ist eine Zuweisung zwischen diesen nötig, müssen entsprechende Anweisungen zum Datenwrappen implementiert werden.

### 7.2.3 Parallelisierung auf höherer Ebene

Die zuvor präsentierten Methoden der Parallelisierung (→7.2.1, 7.2.2) waren bezüglich der Einteilung auf der Ebene 0 anzutreffen und beschränkten sich auf logische Operationen. Aufgrund dessen waren sie nur für die Gatterebene bzw. die Schalterebene geeignet. Arithmetische Operationen, wie man sie auf der Register-Transfer-Ebene (→2.1) antrifft, lassen sich mit zwei- oder vierwertiger Logik mit einer Abbildung auf

1 oder 2 Bits nicht realisieren. Für arithmetische Operationen werden meistens ein ganzes oder sogar mehrere Maschinenwörter verwendet.

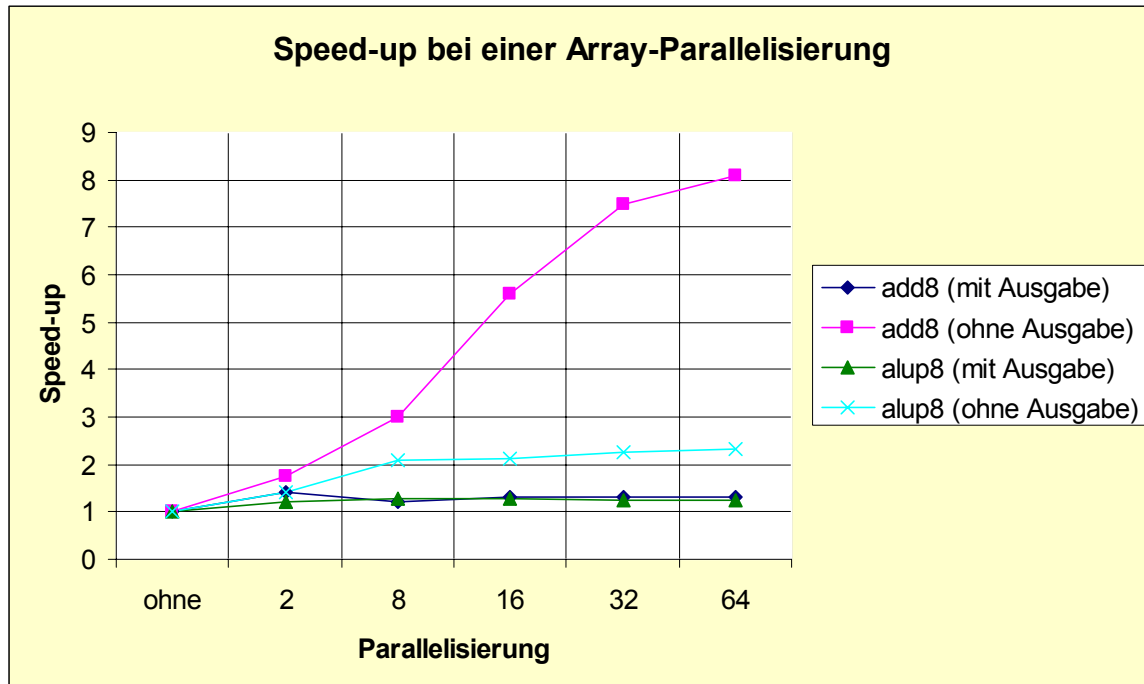
Allerdings lassen sich auch diese Wörter wieder parallelisieren. Hierzu werden zwei Verfahren vorgestellt, die spezielle Mittel von C++ bzw. SystemC nutzen. Eine Übertragung der hier vorgestellten Methoden in andere Hardwarebeschreibungssprachen dürfte nicht einfach bzw. für den parallel-definierten Datentyp unmöglich sein.

### 7.2.3.1 Array-parallele Simulation

Eine mögliche Parallelisierung von Maschinenwörtern ist die in einem Array. Dabei bekommen die Prozesse und Methoden mehrfache Parameter. Zum Beispiel wird aus dem Ausdruck `sc_signal<int> MyDat` ein `sc_signal<int> MyDat[8]`. Statt eines einfachen Integer-Signals gibt es nun acht Signale vom Typ Integer. Um Prozesse zu aktivieren, ist jetzt eine Erweiterung der Sensitivitätsliste auf alle diese acht Signale nötig. Dazu reicht jedoch eine *for*-Schleife, z. B. `for (int i=0; i<8; i++) sensitive << in1[i] << in2[i];`. Ähnlich sieht es bei der Verdrahtung der Module aus. Ein Beispiel hierfür könnte `for (int i=0; i<8 ; i++){ M1.a1[i](SIG_IN1[i]); ...}` sein. Letzten Endes müssen nur noch die implementierten Funktionen in Schleifen ausgeführt werden. Hintergrund dieser Variante ist es, dass in einem Prozessablauf bzw. in einer Methode mehrere Operationen stattfinden und so die Anzahl der Prozessaufrufe minimiert wird. Die Datenparallelität liegt hier in der Kommunikation innerhalb des Programms.

Unter Umständen mögen die hier vorgenommenen Anpassungen etwas umständlich erscheinen, aber es ist ein machbarer Weg. Dieser lässt sich auch automatisieren und der Mehraufwand macht sich bei der Wiederverwendung des gleichen Modells oft schon bezahlt. Einige Simulationsergebnisse für zwei RTL-Modelle in SystemC sind in der Tabelle 7.4 und mit einem zugehörigen Diagramm dargestellt.

Das *add8*-Beispiel ist ein einfaches und die *alup8*-Schaltung ein größeres Modell, welches mehrere Module konkurrent ausführt. Bei den untersuchten Modellen wurde der Einfluss des Parallelisierungsgrades auf den Speed-up näher untersucht. Es ist zu sehen, dass eine Parallelisierung zu einem merklichen Speed-up führt. Allerdings stagniert die Beschleunigung ab einer gewissen Größe des Arrays. Die Datenausgabe wurde so geschrieben, dass sie für alle Array-Größen identisch aussieht. Außerdem sind Ergebnisse aufgeführt, bei denen keine Ausgabe erfolgt. Bei diesen Modellen hat die Modellrechenzeit einen deutlich größeren Einfluss auf die Gesamtrechenzeit. Ohne die Datenausgabe ergibt sich eine deutlich schnellere Ausführung.



Design	Speed-up bei einer Parallelisierung von:					
	ohne	2	8	16	32	64
add8 (mit Ausgabe)	1	1,4	1,22	1,3	1,31	1,33
add8 (ohne Ausgabe)	1	1,75	3	5,6	7,48	8,1
alup8 (mit Ausgabe)	1	1,21	1,28	1,27	1,24	1,25
alup8 (ohne Ausgabe)	1	1,41	2,09	2,14	2,27	2,32

Tabelle 7.4: Simulationsergebnisse für Array-parallele Beispiele

### 7.2.3.2 Simulation eines Parallel-definierten Datentyps

Anstatt wie eben beschrieben, (→7.2.3.1) einen Signaltyp zu vervielfachen, ist es auch möglich einen Signaltyp zu kreieren, welcher aus einem Integer-Array besteht. Hierzu wird eine neue Klasse erschaffen. Die neue Klasse steht für den parallel-definierten Datentyp (PDDT). Im Listing 7.3 ist dies auszugsweise dargestellt.

---

```
class myInt {
2   private:
    int dat[8];
4   public:
    . . .
6   inline myInt & operator = (const myInt & rhs) {
        for (int i=0; i<8; i++) {
8           dat[i] = rhs.dat[i];
        }
10        return *this;
    }
12   inline friend void sc_trace(sc_trace_file *tf, const MyInt & v,
        const sc_string& NAME ) {
14        for (int i=0; i < 8; i++) {
            sc_trace(tf,v.dat[i], NAME + "dat[i]");
16        }
    }
18   . . .
    inline friend MyInt operator + (const MyInt& v, const MyInt& w );
20 };

22 MyInt operator+ (const MyInt& v, const MyInt& w ) {
    MyInt summe;
24    for (int i = 0; i < 8; i++) {
        summe.dat[i] = v.dat[i] + w.dat[i];
26    }
    return summe;
28 }
```

---

Listing 7.3: Parallel-definierter Datentyp

In dieser Klasse sind dann einige Basisoperatoren (z. B. Zeile 6-9) und alle im Modell verwendeten logischen und arithmetischen Operatoren (z. B. Zeile 19 und 22-28) zu überladen. Weitere Methoden, die z. B. das Aufzeichnen von Signalen während der Simulation erlauben, können in dieser Klasse auch ihren Platz finden (Zeilen 12-16).

Das Erstellen einer solchen Klasse kostet zunächst etwas Programmieraufwand, aber einmal erstellt, kann sie beliebig oft in späteren Designs wieder verwendet werden. Ein Vorteil dieser Variante ist, dass das SystemC-Modell nicht so stark modifiziert werden muss, wie bei der Variante mit der Array-Parallelität (→7.2.3.1). Einzig der Austausch des Datentyps ist hierzu nötig, z. B. der Ersatz von `<int>` durch `<myInt>`.

### 7.2.3.3 Vergleich der Wörter-parallelen Simulationen

In der Tabelle 7.5 sind einige Simulationsergebnisse der Beschleunigung für die schon im Abschnitt 7.2.3.1 vorgestellten Modelle dargestellt. Dabei differieren die Ergebnisse für die Array-parallele Variante zum Teil deutlich. Zum einen liegt es daran, dass die Ergebnisse aus einer anderen Messreihe stammen und somit eine gewisse Streuung ihren Einfluss ausübt. Dies wird bei den Ergebnissen für die *alup8* ohne Datenausgabe

deutlich. Bei den Ergebnissen mit Datenausgabe sind die Unterschiede größer. Das liegt daran, dass Datenausgabe in der Tabelle 7.5 so geschrieben wurde, wie sie am leichtesten zu implementieren ging. Bei den Ergebnissen in der Tabelle 7.4 waren die Dateiausgaben dagegen recht umfangreich. Das bedeutet, die Ausgabe bei den parallelen Modellen in diesem Abschnitt ist kompakter als bei der einfachen Variante. Zusätzlich wurde noch untersucht, wie der Speed-up sich ohne Datenausgabe verhält. In diesem Fall hat die Schaltungsberechnung einen großen Einfluss auf die Ausführungszeit. Für diese Modelle zeigt sich, dass der Speed-up ohne Datenausgabe noch einmal zunimmt, insbesondere bei den PDDT-Modellen. Weiterhin ist bei den PDDT-Modellen zu bemerken, dass der Speed-up unabhängig vom Schaltungsmodell eine nahezu konstante Beschleunigung liefert.

Design	1-fach seriell	8-fach Array	8-fach PDDT
add8 (mit Ausgabe)	1	1,96	2,22
add8 (ohne Ausgabe)	1	3,01	5,89
alup8 (mit Ausgabe)	1	1,92	2,14
alup8 (ohne Ausgabe)	1	2,15	5,93

Tabelle 7.5: Speed-up-Simulationsergebnisse für Beispiele höherer Ebene

Welche Parallelisierungsvariante bei den Modellen höherer Ebene zu verwenden ist, hängt in erster Linie vom Anwendungsfall ab. Bezüglich der Ausführungszeit dominiert die Lösung mit dem Parallel-definierten Datentyp. Werden jedoch aus einer externen Bibliothek aufwändige mathematische Funktionen unter Verwendung komplexer Operationen eingesetzt, dann ist die Variante mit den Arrays an Signalen u. U. einfacher zu implementieren. Solche Funktionen sind in der verhaltensorientierten und algorithmischen Designebene durchaus üblich, z. B. mit der Boost-Bibliothek [187].

## 7.3 Beschleunigung durch verteiltes Rechnen

Der Gebrauch von Hochleistungsrechnern mit großen Kapazitäten ist im anspruchsvollen Simulationsbereich nicht ungewöhnlich. Ein großer Nachteil solcher Systeme sind allerdings die hohen Anschaffungs- und Betriebskosten. Seit den 90er Jahren bietet sich aber mit dem Aufkommen leistungsfähiger Arbeitsplatzrechner, deren Vernetzung und der Entwicklung von Standard-Kommunikationssoftware ein neuer Ansatz der Parallelisierung an [76]. Solch ein vernetztes System wird zu einem leistungsfähigen virtuellen Parallelrechner. Der Begriff der Parallelität beschreibt nur allgemein den gleichzeitigen Ablauf von Prozessen. Eine genauere Formulierung der Parallelität eines vernetzten Rechnersystems ist der des *verteilten Rechnens* (engl. *distributed computing*) [77]. Die Verteiltheit kennzeichnet hierbei die räumliche

Trennung der einzelnen Prozesse, welche auf verschiedenen Rechnern ablaufen, die zum Gesamtergebnis beitragen.

In Bezug auf den Parallelisierungsgrad (→Abb. 7.1) ist diese Variante der Ebene 3 zuzuordnen, da sich hierzu die Prozesse auf verschiedenen CPUs befinden. Nach der Einteilung nach Flynn [73] entspricht ein solches Rechnersystem dem MIMD-Modell.

Im Folgenden werden SystemC-Simulationen in einem verteilten System mit Hilfe von MPI vorgestellt. Verteilte Simulationen mit MPI, auch von SystemC-Modellen, sind zahlreich und nicht neu. Die verteilte Simulation von Fehlern findet man dagegen selten. In [90] wird das *Prototyping* eines fehlertoleranten Prozessor in SystemC in einem verteilten System vorgestellt. Die hier vorgestellte Anwendung unterscheidet sich von anderen darin, dass sie eine universelle *Middleware* (MPS) vorstellt, die flexibel an Modelle mit Fehlerinjektion angepasst werden kann.

### 7.3.1 Message Passing Interface (MPI)

Neben der System-Hardware besitzt die Kommunikationssoftware eine entscheidende Rolle. Das am häufigsten verwendete Werkzeug dürfte hierfür das *Message Passing Interface* (MPI) sein [65]. Die Bezeichnung von MPI macht schon deutlich, dass die Kommunikation der Prozesse nachrichtenorientiert erfolgt. Das MPI selbst ist ein vom MPI-Forum herausgegebener Standard der eine einheitliche Schnittstelle definiert [78]. Die Art der Vernetzung, der Rechner oder des Betriebssystems ist hierbei nicht festgelegt. Ohne solche Einschränkungen ist dieser Standard plattformübergreifend. Unter Zuhilfenahme der Bibliotheksfunktionen ist der Anwender von Arbeiten, welche die systemspezifischen Merkmale wie beispielsweise die CPU-Anzahl oder das Netzwerk betreffen, entbunden. So ist es auch möglich, ein MPI-nutzendes Programm auf einem Einzelprozessorsystem in mehreren Prozessen auszuführen. MPI bietet eine gute Skalierbarkeit.

Derzeit ist der Standard MPI-2 aktuell. Bei MPI-2 wurde die Vorgängerversion MPI-1 z. B. um die Möglichkeit paralleler Ein- und Ausgaben und um die Erzeugung dynamischer Prozesse erweitert. In dieser Arbeit wird das MPI-2 verwendet.

Die Umsetzung des MPI-Standards erfolgt in Form von Bibliotheken. Ein kommerzielles Werkzeug ist z. B. das Intel Cluster Toolkit [79]. Neben kommerziellen Bibliotheken gibt es auch eine Reihe von frei verfügbaren. Besonders bekannt ist das MPICH [80] und das LAM-MPI [81]. MPI-Portierungen gibt es aber nicht nur für verschiedene Rechnerplattformen und Betriebssystemen, sondern auch für diverse Programmiersprachen wie z. B. auch für Fortran [82]. Eine MPI-Umgebung ist vorzugsweise für homogene Systeme und zur Erzielung einer maximalen Performance ausgelegt. Eine andere, weit verbreitete Kommunikationssoftware ist das *Parallel Virtual Machine* (PVM) [83]. Diese Umgebung ist eher für heterogene Systeme gedacht und je nach Situation nicht so leistungsfähig wie das MPI. Soll jedoch eine MPI-Applikation in heterogenen Systemen stattfinden, so ist ein erheblicher Aufwand zur



Interprozesskommunikation nötig, welcher sich auch durch eine Einbuße an Geschwindigkeit ausdrückt.

Mit mehr als 100 Befehlen von Kommunikations- und Gruppenfunktionen bietet sich MPI als leistungsfähiges Werkzeug an. Typische Befehle, die sich in nahezu jeder MPI-gestrickten Applikation wiederfinden, sind die Initialisierung, die Zuordnung von Prozessen, das Senden und Empfangen, das Synchronisieren oder das Beenden. Typisch für MPI-Funktionen ist, dass sie mit dem Präfix *MPI\_* beginnen.

### 7.3.2 Aufbau des Rechnersystems

Zur Verwendung der Simulation des verteilten Rechnens wird eine Gruppe von PCs verwendet, die über ein Standard-Netzwerk (Ethernet) miteinander kommunizieren können. Eine solche Anordnung wird auch als PC-Cluster bezeichnet. Die Abbildung 7.2 gibt den Aufbau visuell wieder.

Mittelpunkt des Systems bildet ein PC, der als *Master* fungiert. Die verbleibenden Rechner im Netz sind *Slaves*. Während der Master-PC sich für verwaltende Arbeiten verantwortlich zeichnet, finden auf den Slave-PCs die Berechnungen statt. Die Kommunikation erfolgt als Punkt-zu-Punkt-Variante zwischen dem Master und den Slaves. Ein Informationsaustausch der Slaves untereinander ist nicht erforderlich. Eine Simulation kann über ein Netzwerk von einem beliebigen PC initiiert werden. Dieser Rechner (Remote-PC) muss dem PC-Cluster nicht zugehörig sein.

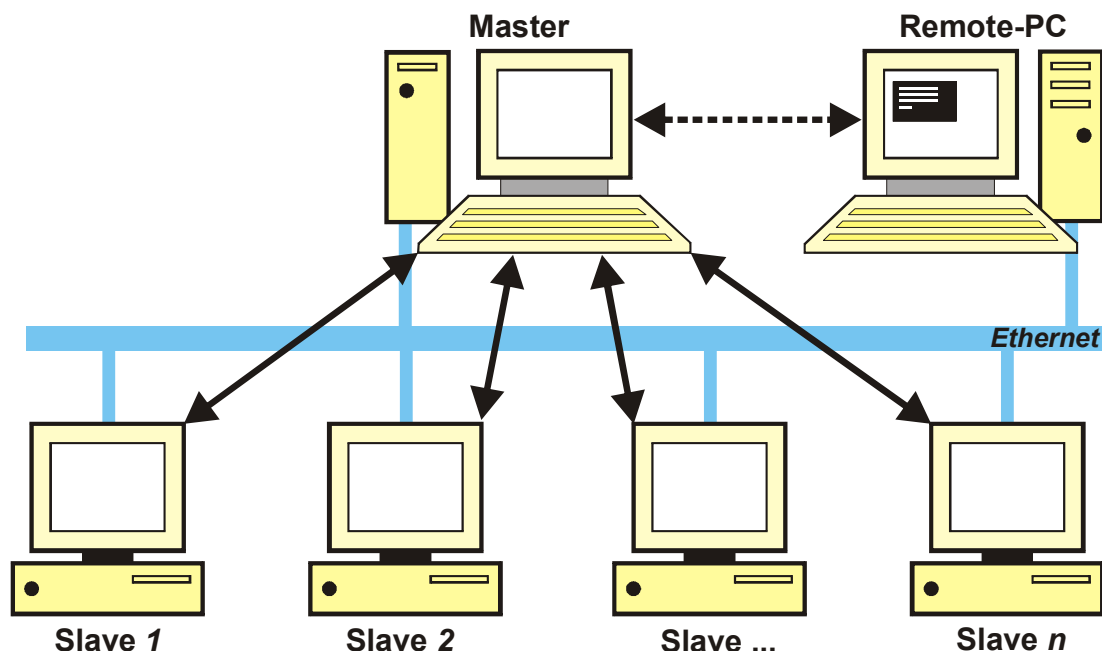


Abbildung 7.2: Aufbau des PC-Clusters

Da ein homogenes System den Einsatz von MPI erleichtert und verbessert, sind einige Voraussetzungen bezüglich der Architektur, des Betriebssystems und des Netzwerkes zu erfüllen. Bei den verwendeten Rechnern handelte es sich um IBM-kompatible PCs mit ähnlicher HW-Ausstattung. Das Netzwerk war ein gewöhnliches Fast-Ethernet-Netzwerk mit 100 MBit/s. Für das Betriebssystem wurde Linux, ein von Linus Torvalds ursprünglich entwickeltes UNIX-System [84], verwendet. Die Abbildung 7.3 zeigt die Installation aus der Software-Sicht.

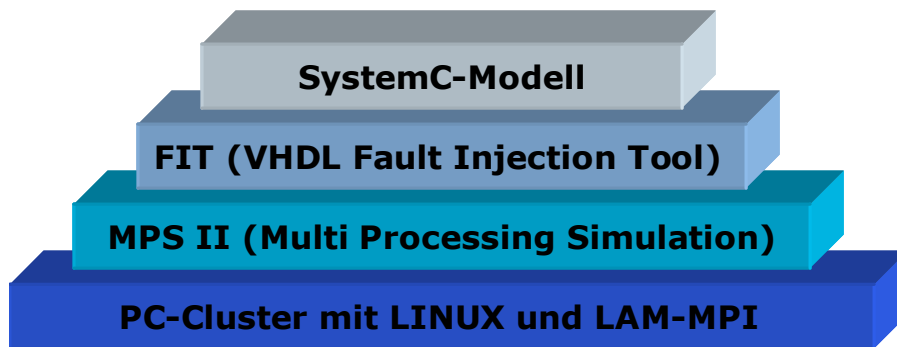


Abbildung 7.3: Schichtenbasiertes Software-Modell

Grundlage bildet eine Linux-Distribution [85] auf der das LAM-MPI [81] installiert ist. In der LAM-MPI-Umgebung läuft das MPS, welches den Simulator von den Aufgaben der Parallelisierung entbindet. Als Schnittstelle zwischen dem SystemC-Modell und dem MPS kommt der FIT-Simulator zum Einsatz. Die Verwendung des FIT zur verteilten Simulation bietet sich an, da eine Mixed-Language-Co-Simulation zu keiner Leitungseinbuße führt (→6.5). Andererseits sind im FIT schon die Funktionen zur Kommunikation mit dem MPS integriert. Ohne die Verwendung des FIT müsste für jedes SystemC-Projekt eine erneute Implementierung der Kommunikationsfunktionen erfolgen.

### 7.3.3 Multi-Processing-Simulation (MPS)

Inspiziert von einer Arbeit [86], welche durch ein Zusatzprogramm eine verteilte Simulation mittels PVM [83] für heterogene Systeme realisiert, entstand das Multi-Processing-Simulation (MPS). Das MPS erlaubt es, aus einer Einzelplatzsimulation eine verteilte Simulation in einem PC-Cluster zu betreiben. Bei einer typischen Simulation ist jeder PC im Netz ein Knoten auf dem genau ein Simulationsprozess abläuft, ausgenommen ist der Master-Knoten. Voraussetzung für ein erfolgreiches verteiltes Berechnen eines Problems bildet die Zerlegung der Aufgabe. Für die Simulation von Fehlern bietet sich hierfür die Partitionierung der Fehlerliste an. Es wird die Datenparallelität genutzt. In der Abbildung 7.4 ist das verwendete Schema dargestellt.

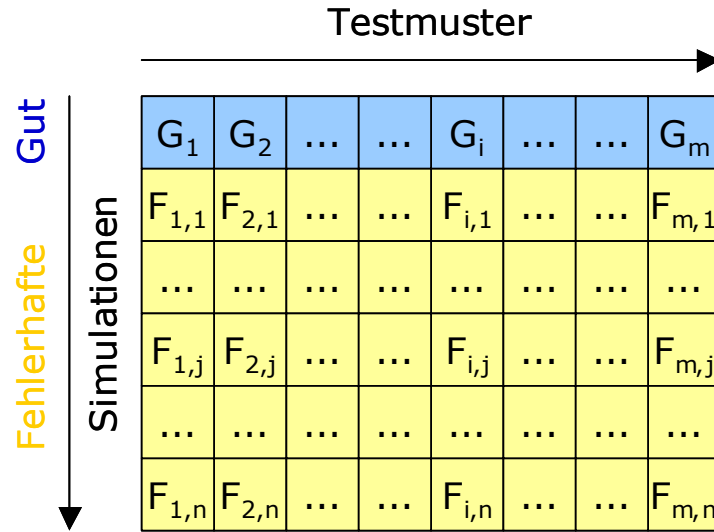


Abbildung 7.4: Ablauf für Simulationen mit mehreren Einzelfehlern

Bei einer funktionalen Simulation, auch als Gut-Simulation bezeichnet, werden  $m$ -Testmuster  $G_i$  sequenziell berechnet. In der Abbildung 7.4 entspricht dies der ersten Zeile. Für die Simulation eines Fehlers wird zunächst ein Fehler  $j$  injiziert und dann für alle  $m$ -Muster  $F$  berechnet. Anschließend erfolgt die Simulation der nächsten Fehler, bis alle Fehler  $n$  abgearbeitet sind. Die Anzahl aller Simulationsschritte  $A_{ss}$  ergibt sich somit aus:

$$A_{ss} = \sum_{i=1, j=1}^{i=m, j=n} F_{i,j} + \sum_{i=1}^{i=m} G_i \quad (7.12)$$

Die Zerlegung besteht nun darin, dass die Gut-Simulation und jede Simulation mit einem Fehler ein unabhängiges Simulationsmodell bildet. Für die Gut-Simulation gilt für die Anzahl der Simulationsschritte  $A_{SSG}$ :

$$A_{SSG} = \sum_{i=1}^{i=m} G_i \quad (7.13)$$

Für eine Simulation mit der Fehlerinjektion  $F$  ist die Anzahl der Simulationsschritte  $A_{SSF}$  entsprechend:

$$A_{SSF} = \sum_{i=1}^{i=m} F_{i,j} \quad (7.14)$$

Die Hauptaufgabe des MPS besteht darin, die verschiedenen Simulationsmodelle effizient auf die Schaltungsknoten zu verteilen. Je nach Zuteilung ist das der MPS-Master- oder Slave-Knoten.

Folgende Funktionalitäten realisiert das MPS im Detail:

- **Verteilen der Simulationsprojekte:** Jeder PC bekommt die gleiche Projektdatei. Im Projekt befindet sich das Schaltungsmodell, die Muster- und Fehlerliste, die Konfigurationsdateien und zumeist auch der Simulator.
- **Packen/Entpacken:** Für eine schnelle Kommunikation werden die Daten in gepackter Form übertragen. Das MPS kümmert sich um das Packen und Entpacken der zu übertragenden Dateien und Informationen.
- **Das Anlegen der Simulationsumgebung:** Für eine Simulation ist es z.T. notwendig, Unterverzeichnisse oder temporäre Ordner anzulegen.
- **Simulationsstart:** Die Aufforderung eine Simulation zu starten, wird vom MPS initiiert. Außerdem bekommt der Simulator die Mitteilung, welcher Fehler injiziert werden soll. Das Simulationsende ergibt sich im regulären Verlauf automatisch durch das Erreichen des Endes der Datei mit den Eingabemustern.
- **Ablegen der Simulationsergebnisse:** Die Ergebnisse der verteilten Simulationen werden in einem zentralen Ordner zur späteren Auswertung gesammelt.
- **Fehlerbehandlung:** Das verteilte System ist nicht vor dem Auftreten irregulärer Zustände sicher. Während der Simulation können sowohl Fehler auf den lokalen Knoten, wie z. B. Speicherüberläufe, als auch Fehler in der Kommunikation auftreten. Damit diese Fehler nicht zu Deadlocks in der Gesamtsimulation führen, sind im MPS Fehlerbehandlungsroutinen vorhanden.

Um die vorgestellten Funktionalitäten zu erfüllen, besteht die MPS-Anwendung aus einer Reihe von Klassen. Die Wichtigsten kommen kurz zur Erläuterung. Je nach Aufgabe wird im MPS ein Objekt der Master- oder Slave-Klasse erzeugt. Um die Verteilung der Fehlerinjektionen auf die Knoten vorzunehmen, existiert eine *Fault-Manager*-Klasse und bei der Verwaltung der verschiedenen Knoten selbst hilft der *Rank-Manager*. Unterstützt wird die Simulation durch diverse Konfigurationsdateien, für die es eine *Config-File-Manager*-Klasse gibt. Das MPS-Klassendiagramm befindet sich mit Erläuterungen im Anhang A.4.

Die Kommunikation zwischen dem Schaltungssimulator und dem MPS geschieht mittels Pipe-Funktionen des Linux-Systems [87]. Eine Pipe sorgt dafür, dass die Ausgabe des einen Prozesses in die Eingabe eines anderen gelangt.

### 7.3.4 Ablauf der verteilten Simulation

Voraussetzung für die Simulationen ist, dass auf jedem PC das LAM-MPI und das MPS installiert sind. Nachdem das LAM-MPI als Demon-Prozess gestartet wurde, lässt sich das MPS unter Angabe diverser Parameter ausführen. Ein wichtiger Parameter ist zum Beispiel die *Host*-Liste, in der die IP-Adressen der zu verwendenden PCs stehen. Ist ein Knoten der erste Prozess der Gesamtsimulation, so richtet MPS diesen als *Master*-Knoten ein. Die verbleibenden Knoten haben dann alle *Slave*-Prozesse.

Die Abbildung 7.5 zeigt den Programmablaufplan für den Masterprozess. Für den Masterprozess existiert eine Konfigurationsdatei, in der das zu verwendende Design, die Ergebnisausgabe und die Form der zu schreibenden Protokollierungen stehen. Anhand dieser Konfigurationsdatei wird der Masterprozess initialisiert. Anschließend erfolgt die Verteilung der Designs an die Slave-Knoten.

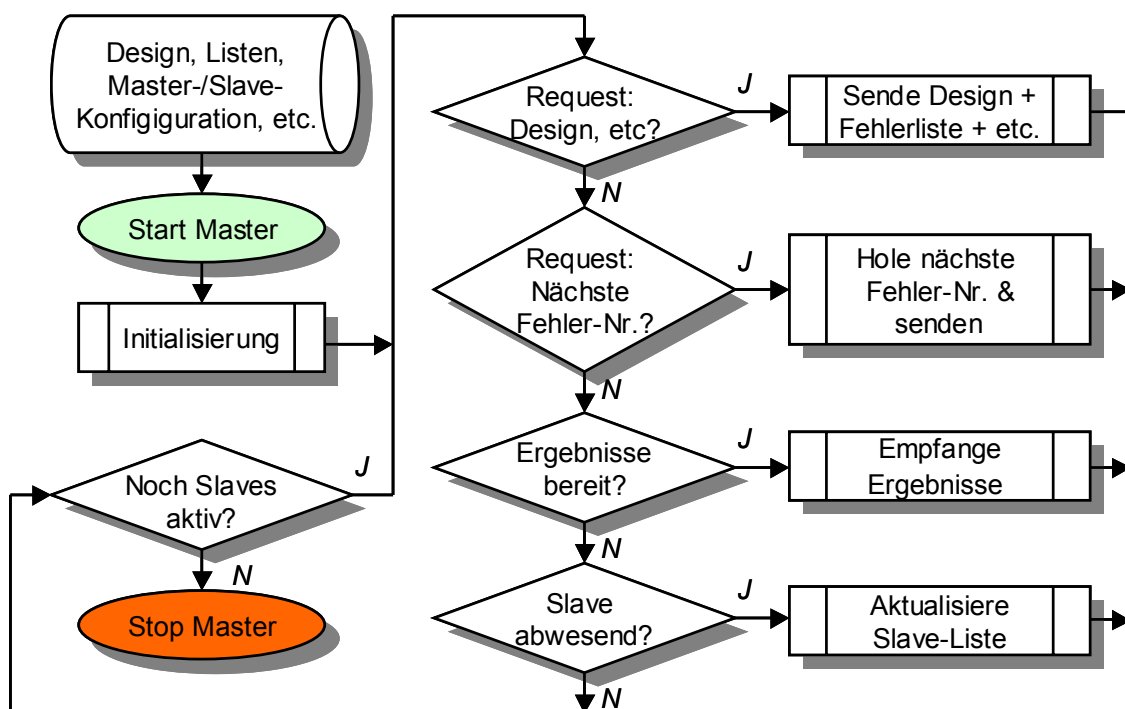


Abbildung 7.5: Programmablaufplan für den Master-Prozess

Für die zu simulierenden Fehler wird auf dem Master eine Aufzählungsliste geführt. Anhand dieser Liste weiß der Master, welche Fehler schon simuliert wurden, welche in Bearbeitung sind und welche noch ausgeführt werden müssen. Erfordert ein Slave zur Simulation eine gültige Fehlernummer, so wird aus dieser Aufzählungsliste eine Fehlernummer gesendet. Jeder Slave besitzt die vollständige Fehlerliste und kann anhand der empfangenen Nummer den zu injizierenden Fehler zuordnen. In der Praxis

hat sich gezeigt, dass die Verwaltung und Kommunikation per Fehlernummer etwas effizienter ist, als einen vollständigen Eintrag aus der Fehlerliste zu senden. Ist ein Slave mit einer Simulation fertig, so empfängt der Master die Ergebnisse und legt sie laut Konfiguration ab. Ist ein Slave, ganz gleich aus welchen Gründen, nach einem Time-out nicht mehr erreichbar, so wird dieser aus der Liste der zur Verfügung stehenden Rechner gestrichen. Er wird für die verbleibenden Simulationen nicht mehr berücksichtigt. Ist überhaupt kein Slave-Prozess mehr aktiv, wird die gesamte Projektsimulation beendet. Für den Slave-Prozess ist der Programmablauf in der Abbildung 7.6 dargestellt.

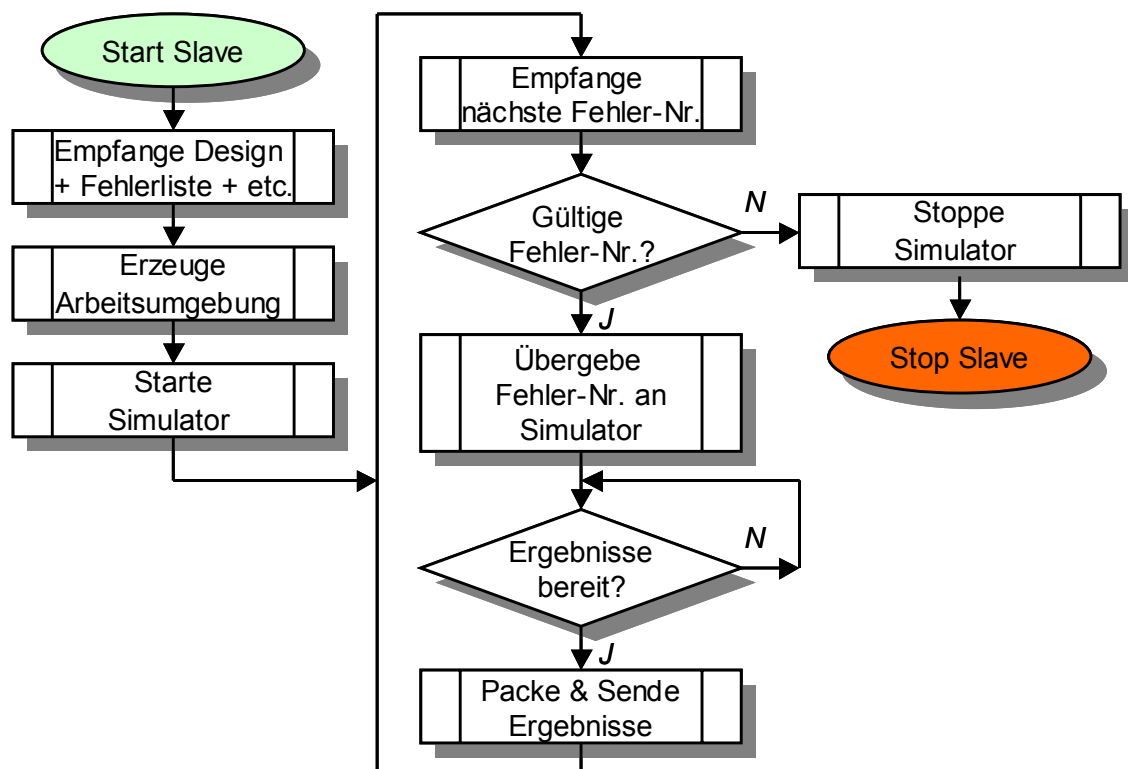


Abbildung 7.6: Programmablaufplan für den Slave-Prozess

Nachdem der Slave das Design vom Master empfangen hat, erzeugt er die Rahmenbedingungen für die Simulation gemäß einer Konfigurationsdatei. Nach dem Start des Simulators wird eine vom Master zu sendende, gültige Fehlernummer erwartet. Ist die Fehlernummer oberhalb des gültigen Bereiches, ist dies als Aufforderung zum Beenden der Simulation zu werten. Hierzu werden zunächst der Simulator und anschließend der MPS-Slave-Prozess gestoppt. War die übergebene Fehlernummer gültig, wird diese an die Fehlersimulation weitergereicht. Nachdem der Simulator entsprechend der Fehlernummer den Eintrag aus der Fehlerliste selektiert und mit ihm einen Eingabesatz berechnet hat, wird dem Master signalisiert, dass die Ergebnisse vorliegen. Der Slave komprimiert die Ergebnisse, löscht die temporären Ausgaben und sendet die gepackten Ausgaben zum Master. Diese Simulationsschritte

wiederholen sich, bis der Master durch eine ungültige Fehlernummer zum Beenden der Simulation auffordert.

Das Problem des Lastausgleichs ließ sich durch die Partitionierung der Fehlerliste gut lösen. Sobald der Master einen freien Slave bemerkt, bekommt dieser anhand der Fehlernummernliste eine neue Aufgabe zugeteilt. Um die Verwaltung und das Einsortieren der Ergebnisse kümmert sich der Master. Auf Wunsch lässt sich per Konfigurationsdatei der detaillierte Ablauf des verteilten Rechnens protokollieren.

### 7.3.5 Beispielsimulationen

Um eine Simulation eines SystemC-Modells mit Fehlerinjektion auszuführen, wurde wie bereits oben erwähnt die Mixed-Language-Co-Simulation mit dem FIT-Simulator verwendet. Dieses Vorgehen entbindet die SystemC-Simulation von der Kommunikation mit dem MPS. Allerdings besteht nun das Problem, dass das FIT keine Fehlerinjektion in eine HLD-Komponente erlaubt. Es weiß nichts über ihre innere Zusammensetzung, im Gegensatz zu einer VHDL-Beschreibung. Ein Zugriff ist einzig über die Parameter der dedizierten Funktionen erlaubt (→6.1). In [223] wurde das Problem so gelöst, dass das Modell um zusätzliche Parameter in Form weiterer Eingänge erweitert wurde. Über diese Eingänge sind die Saboteure im SystemC-Modell ansprechbar (→5.4.1). Die Aktivierung der Fehlerinjektion erfolgte dabei nicht über die Fehlerliste, wie bei der sonstigen VHDL-Injektion, sondern mittels zusätzlicher Einträge in der Datei mit den Eingabemustern. Für die verteilte Simulation ließ sich diese Lösung nicht nutzen, da das MPS nur Einträge aus der Fehlerliste partitionieren und sinnvoll verteilen kann. Ein Partitionieren der Datei mit den Eingabemustern durch das MPS würde nicht nur die Simulationsvorbereitung verkomplizieren, es wären auch zusätzliche Eingriffe zur Definition von Nebenbedingungen durch den Nutzer erforderlich.

In den hier verwendeten verteilten Simulationen sieht die Lösung des Problems so aus, dass als Injektionshilfe ein einfaches VHDL-Modell in das Simulationsprojekt eingefügt wurde. In der Abbildung 7.7 ist dies schematisch dargestellt. Das VHDL-Modell besteht im Wesentlichen nur aus einfachen Buffern, in welche das FIT versucht Fehler zu aktivieren. Diese Fehlerinjektionen werden aber vom VHDL-Modell über zusätzliche Ports an das SystemC-Modell weiter geleitet. Die wirksame Fehlerinjektion erfolgt über die SystemC-Testbench in das SystemC-Modell (→5.4).

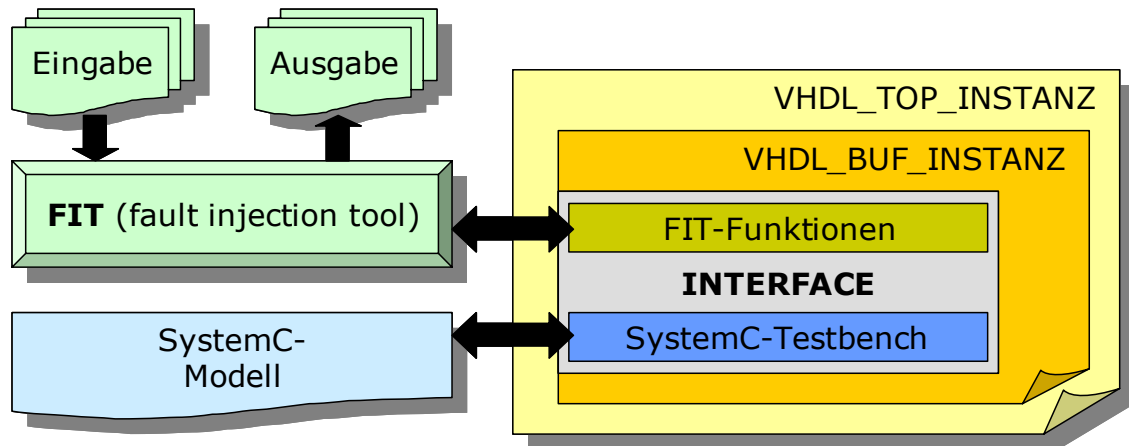


Abbildung 7.7: Schema der Simulationsprojekte für die verteilte Simulation

Bei einer verteilten Simulation sind in dieser Arbeit bezüglich der Effizienz zwei Extreme denkbar. Zum einen liegt der Anteil an Modellrechenzeit weit unter dem Kommunikationsaufwand. Zum anderen ist der Kommunikationsaufwand im Vergleich zur Rechenzeit sehr gering. Für den ersten Fall dürfte die Effizienz weit geringer ausfallen als für den zweiten. Dieser Umstand wird ausgehend aus dem Amdahlschen Gesetz ( $\rightarrow$  Formel 7.7) in der Formel 7.15 ausgedrückt:

$$S_p(P) = \frac{1}{f + K(P) + \frac{1-f}{P}} \leq \frac{1}{f} \quad (7.15)$$

Der Anteil der Kommunikation wird durch  $K$  gebildet, welcher natürlich von der Anzahl der beteiligten Prozesse abhängt. Bildet man den Grenzwert für den Speed-up, so ergibt sich:

$$\lim_{P \rightarrow \infty} S_p(P) = \lim_{P \rightarrow \infty} \frac{1}{f + K(P) + \frac{1-f}{P}} = \frac{1}{f + K(P)} \quad (7.16)$$

Unter der Annahme, dass  $f \ll K(P)$  ist, gilt vereinfacht:

$$\lim_{P \rightarrow \infty} S_p(P) = \lim_{P \rightarrow \infty} \frac{1}{f + K(P) + \frac{1-f}{P}} = \frac{1}{K(P)}; (f \ll K(P)) \quad (7.17)$$

Der nicht zu verachtende Zeitanteil des Packens und Entpackens des Designs bzw. der Ergebnisse bei einem MPS-Projekt ( $\rightarrow$  7.3.3, 7.3.4) wird der Berechnungsausführung und nicht der Kommunikation zugerechnet. Jedoch beeinflusst



diese Maßnahme den Anteil der Kommunikation, da die Datenpakete kleiner werden, sinkt der Kommunikationsanteil.

Der Einfluss des Schaltungsmodells auf die Skalierbarkeit der verteilten Simulation wurde anhand einiger ISCAS-Benchmark-Schaltungen [88,89] näher untersucht. Das PC-Cluster war ein Pool von Laborrechnern, welche Athlon-CPU's mit einer Taktfrequenz von 1 GHz und einen RAM-Speicher von jeweils 512 MB hatten. Die Vernetzung erfolgte über ein Fast-Ethernet-Netzwerk.

Die Abbildung 7.8 zeigt ein Diagramm, in dem einige Simulationsergebnisse für die Speed-ups einiger ISCAS-Schaltungen dargestellt sind.

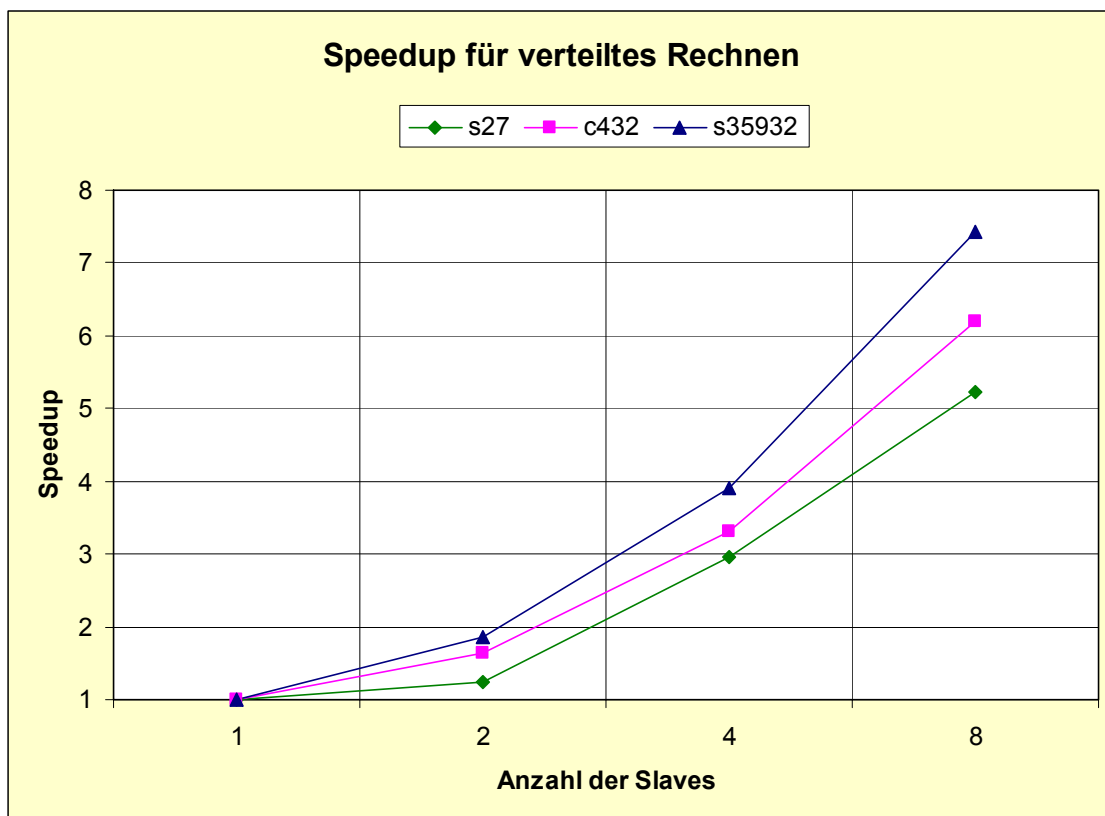


Abbildung 7.8: Speed-up für einige ISCAS-Schaltungen

Es zeigt sich, dass die kleinste Schaltung *s27* die schlechtesten Werte bezüglich des Speed-up aufweist. In diesem Modell ist der Anteil des Datenaustausches mit dem Master-Prozess im Vergleich zur Simulationsausführung recht hoch. Für die Schaltung mittlerer Größe *c423* findet man deutlich bessere Werte vor. Die größte Schaltung *s35932* weist die besten Werte bezüglich der Beschleunigung auf. Der Speed-up skaliert recht gut mit der Anzahl der Slaves-PCs. Hier ist der Zeitanteil der Schaltungsberechnung deutlich höher als der für den Datenaustausch. Es zeigt sich, dass das Verfahren der verteilten Simulation sich gerade für simulationskritische Anwendungen bezüglich der Ausführungszeit gut eignet.

Ab einem gewissen Punkt dürfte die Kommunikation am Master-PC zum Flaschenhals werden. Experimentell konnte dies nicht ermittelt werden, da nicht genügend Hardware zur Verfügung stand. Es wird aber eingeschätzt, dass eine Rechneranzahl von mehr als 100 in den seltensten Fällen noch zu einer Beschleunigung führen dürfte. Eine Weitung des Engpasses erreicht man durch einen Hardware-Ausbau am Master-PC. Dabei dürfte der Einsatz einer zweiten Netzwerkkarte das einfachste Mittel sein. Prinzipiell lässt sich das System weiter skalieren, wenn mehrere MPS-Projekte aufgesetzt werden und nicht wie hier vorgestellt, nur eines zum Einsatz kommt. Mehrere MPS-Projekte können entweder völlig unabhängig nebeneher ausgeführt werden. In diesem Fall ist eine vorherige Partitionierung der Fehlerliste sinnvoll. Andererseits kann auch eine hierarchische Mastermodellierung implementiert werden. In diesem Fall kommt eine Verwaltung der Master einzelner MPS-Projekte von übergeordneten Mastern zum Einsatz. Jedoch müssen hierzu für die übergeordneten Master die entsprechenden Klassen in der MPS-Applikation erweitert bzw. überschrieben werden.

### 7.4 Fazit des beschleunigten Rechnens

In den Abschnitten 7.3 und 7.4 wurden verschiedene Methoden der Parallelisierung zum Zwecke einer Simulationsbeschleunigung vorgestellt. Bei den Bit-parallelen Techniken wird die Parallelisierung über das Maschinenwort realisiert. Somit wird die Skalierbarkeit durch dessen Breite beschränkt. Ebenso nutzt das Verfahren des selbstdefinierten Datentyps von den vierwertigen Methoden die Breite eines Bytes. Der SystemC-Logikvektor darf zwar größer sein, aber dessen Maximallänge ist auch beschränkt. Bei dem Array-parallelen und dem PDDT-Verfahren ist bezüglich der Skalierbarkeit keine enge Grenze gesetzt. Allerdings zeigte sich, dass sich ab einer bestimmten Größe keine Geschwindigkeitsvorteile ergeben. Der Weg des verteilten Rechnens zeigt die größten Reserven für eine schnellere Simulation, allerdings mit zusätzlichen HW-Kosten.

Eine weitere Möglichkeit der Beschleunigung liegt in der Kombination der vorgestellten Methoden. In der Tabelle 7.6 ist zur besseren Übersicht die Kombination von jeweils zwei Methoden dargestellt. Bei einer umfassenden Betrachtung lassen sich auch mehr als zwei Verfahren kombinieren. Aus der Tabelle kann man aber schlussfolgern, dass wenn *Verfahren 1* und *Verfahren 2* sich miteinander kombinieren lassen und beide auch mit einem *Verfahren 3* funktionieren, so sind alle drei Varianten in einem Modell möglich. Die Eintragungen haben folgende Bedeutungen:

- +: diese Kombination ist möglich,
- -: diese Kombination ist nicht möglich,
- x: diese Kombination ist mit der jetzigen Implementierung noch nicht möglich, aber mit Modifikationen realisierbar.

Methoden	MP	FP	sc_lv	sc_sd	AP	PDDT	VR
VR	x	x	x	x	x	x	
PDDT	+	+	+	+	-		
AP	+	+	+	+			
sc_sd	-	-	-				
sc_lv	-	-					
FP	-						
MP							

Tabelle 7.6: Kombination paralleler Verfahren

Die Kürzel entsprechen den folgenden parallelen Methoden:

- MP: musterparallele Beschleunigung (→7.2.1.1),
- FP: fehlerparallele Beschleunigung (→7.2.1.2),
- sc\_lv: Variante mit dem SystemC-Logikvektor (→7.2.2.1),
- sc\_sd: Methode mit dem selbstdefinierten Datentyp (→7.2.2.2),
- AP: Array-paralleles Verfahren (→7.2.3.1),
- PDDT: Methode mit dem parallel-definierten Datentyp (→7.2.3.2),
- VR: Beschleunigung durch verteiltes Rechnen (→7.3).

Die Einschränkungen in der Parallelisierung ergeben sich z. B. dadurch, dass zwei Verfahren die gleiche Ressource nutzen. Dies ist beispielsweise bei der fehlerparallelen und der musterparallelen Variante der Fall.

Rein theoretisch ergibt sich für den Speed-up eine Multiplikation aus den einzelnen Werten. Jedoch kann die gleichzeitige Simulation mehrerer unterschiedlicher Werte für den ereignisgesteuerten Simulator auch zu mehr Aktivität führen, was den theoretischen Leistungsgewinn wieder schmälert.



## 8 Zusammenfassung und Ausblick

*Schreiben ist hart;  
man kommt nur schwer dahinter,  
wann man aufhören muss.  
Peter Ustinov (1921-2004),  
engl. Schriftsteller u. Schauspieler*

Im letzten Kapitel werden die Ergebnisse der Arbeit kurz zusammengefasst und ein Überblick zu den wichtigsten Bestandteilen gegeben. Ein Ausblick auf weiterführende Ideen und Ansätze bildet den Abschluss.

In dieser Arbeit wurden verschiedene Strategien zur Simulation von HW-Systemen mit Fehlern vorgestellt. Die Hilfsmittel zur Modellbeschreibung sowie zur Simulation waren die SystemC-Bibliothek (2.0.1) und diverse C++-Compiler. Gründe für eine Simulation mit Fehlern können verschiedene Ursachen haben. Wesentliche Motive sind die Qualitätsüberprüfungen von Eingabe-Pattern für den Fertigungstest und der Verhaltensnachweis für ein System bei auftretenden Fehlern im Betrieb. Für diese Arbeit war das Einsatzziel der Simulation primär unerheblich. Es wurden zunächst verschiedene Techniken untersucht und deren Implementierung in SystemC realisiert. Es zeigte sich, dass solche Techniken wie die der Saboteure und Mutanten, welche schon aus Fehlerinjektionsexperimenten mit VHDL bekannt waren, auch in SystemC realisiert werden konnten. Vorteile, die auf dem C++-Erbe beruhen, kamen z. B. zusätzlich bei den parametrisierten Modulen zum Tragen. Die generischen SystemC-Saboteure und -Mutanten erwiesen sich als flexibler als ihre Entsprechungen in anderen Hardwarebeschreibungssprachen. Ebenso bilden SystemC-Komponenten gute Voraussetzungen für Anbieter von IPs.

Für Fehlerinjektionen, bei denen keine Umsetzung in SystemC möglich war, konnten mit Änderungen in der SystemC-Bibliothek Ersatzlösungen geschaffen werden. Diese neuen Verfahren, welche als Simulationskommandos bezeichnet wurden, erwiesen sich als sehr effizient und konnten mit den anderen Techniken kombiniert werden. Bei der Erweiterung der SystemC-Bibliothek wurde derart Rücksicht genommen, dass andere SystemC-Modelle ohne Fehlerinjektionen ebenso mit dieser Bibliothek ohne Einschränkungen und Effizienzverluste arbeiten können.

Neben dem klassischen Haftfehlermodell kam es zur Implementierung weiterer Fehlermodelle, um mehr realitätsnahe Defekte und Störungen simulieren zu können. Während der erste SystemC-Simulationskernel noch zyklenbasiert war, konnten mit den Erweiterungen der Ereignissteuerung bei nachfolgenden SystemC-Versionen sogar komplexere Verzögerungsmodelle erstellt werden.

Bei der Wahl der zu simulierenden Abstraktionsebene zeigte sich im Verlauf der Arbeit, dass selbst die Gatterebene zur Modellierung diverser HW-Bauteile zu abstrakt und ungenau war. Mit Hilfe von Erweiterungen und zusätzlichen Modulen wurde eine neue Modellierungsebene für SystemC eingeführt. Mit Hilfe dieser Schalterebene entstand die Möglichkeit einer Modellbildung und Simulation, die deutlich näher die Realität widerspiegelt als die bisherigen Entwurfsebenen. So konnten Technologiebedingte Besonderheiten berücksichtigt werden, die auf der Gatterebene schwer nachzubilden sind. Speziell für die Simulationen mit Fehlerinjektion ergaben sich genauere Ergebnisse. Der Mehraufwand an Simulationszeit lag dabei für die untersuchten Schaltungen in einem vertretbaren Rahmen. Entwurfsbereiche, die bisher den Sprachen VHDL und insbesondere Verilog vorbehalten waren, konnten mit SystemC-Beschreibungen nun auch praktikabel modelliert werden.

Die Anbindung von simulierbaren SystemC-Modellen mit einer anderen Simulationsanwendung war ebenfalls Gegenstand der Arbeit. Die vorgestellte Thread-Implementierung erwies sich dabei als sehr effizient im Vergleich zu anderen Co-Simulationen. Mit der Co-Simulation von SystemC und VHDL sind z. B. HW-Modelle aus früheren Entwürfen in ein neues Projekt übertragbar, ohne dass eine Konvertierung oder ein Re-Design nötig wird. Der vorgestellte Ansatz ist aber nicht auf die präsentierte Anwendung beschränkt. Mit dem vorgeschlagenen Weg des Interfaces lassen sich auch andere Programme an eine SystemC-Simulation koppeln. So ist es beispielsweise denkbar, ein Systemmodell mit einer grafischen Oberfläche (GUI) zu versehen, um eine leicht bedienbare Interaktion der Simulation mit dem Anwender zu erreichen. Aufbauend auf ein Prozessormodell kann so ein leicht bedienbarer Instruction Set Simulator entstehen.

Eine schnelle Ausführung ist ein wesentliches Verkaufsargument für einen Simulator. Der Wunsch, eine gestellte Aufgabe möglichst effizient zu lösen, ist und bleibt eines der primären Ziele. Im Rahmen dieser Arbeit wurden auch verschiedene Wege der Simulationsbeschleunigung untersucht. Hierbei lieferten fast alle vorgestellten Techniken von Parallelisierungen und verteiltem Rechnen brauchbare Ergebnisse und zum Teil deutliche Steigerungen. Anhand einiger Beispielsimulationen wurde dies nachgewiesen. Es ließen sich aber auch einige Grenzen der Skalierbarkeit bei der Parallelisierung feststellen.

In dieser Arbeit wurden viele Aspekte, die im Kontext einer Simulation stehen, näher beleuchtet. Dazu gehörten die HW- und Fehlermodellierung, Fehlerinjektionstechniken, Co-Simulation und paralleles Rechnen. Jeder dieser Bereiche stellt allein für sich ein Fachgebiet dar, in dem viele Menschen tätig sind. Viele dieser Ingenieure und

---

Wissenschaftler tauschen sich über Zeitschriften, Bücher, Konferenzen und dem Internet aus. Unter diesem Gesichtspunkt erweist sich eine erschöpfende Untersuchung des hier aufgeführten Themas schon aus zeitlichen Gründen als nicht machbar. Es bleibt auch weiterhin noch viel Platz für weitere Ideen und Ansätze.

Aus einigen Publikationen sind SystemC-Simulationsbeispiele bekannt, die sich auf abstrakteren Entwurfsebenen bewegen, als die hier behandelten. Ein wesentlicher Sinn solcher Modellierungen ist z. B. die Erhöhung der Ausführungsgeschwindigkeit mit der Inkaufnahme des Verlustes an Genauigkeit. Für solche High-Level-Modelle wäre eine systematische Untersuchung verschiedener Fehlerinjektionsverfahren, z. B. auch der in dieser Arbeit präsentierten, nützlich. Zusätzlich wurden eine Reihe von Fehlermodellen und Hardwarekomponenten und deren Verschaltungen in SystemC präsentiert. Neben diesen gibt es noch weitere, so dass auch hierfür noch Modellierungsbedarf besteht. Außerdem entstehen mit dem Aufkommen neuer und der Verbesserung bestehender Technologien bisher unbekannte Effekte, für die sich eine künftige SystemC-Modellierung lohnen wird.

Bei der Parallelisierung wurden einerseits Techniken verwendet, die größtenteils auf einer niedrigen Ebene der Parallelisierung angesiedelt waren. Für diese Verfahren genügte ein Ein-Prozessor-System mit einem gemeinsamen Speicherbereich. Andererseits wurde eine Parallelisierung auf einem Mehrrechnersystem vorgestellt. Hierbei kommen mehrere Prozessoren mit separaten Speicherbereichen zum Einsatz. Neben diesen beiden Ansätzen existiert noch das Modell des Mehrprozessorsystems mit einem gemeinsamen Speicher. Solche Systeme wurden in letzter Zeit populär und besitzen sogenannte Multicore-Prozessoren. In ersten Experimenten konnte mit einem autoparallelisierenden Compiler und den Parallelverfahren mit Schleifenausführungen eine weitere Leistungssteigerung gemessen werden. Hierbei wurde die Option von OpenMP [67] gewählt. Es bedarf jedoch in diesem Bereich noch weiterer Anstrengungen, um bessere Ergebnisse zu erreichen. Mit dem Aufkommen besserer Compiler und Prozessoren mit mehr als zwei Kernen dürfte sich hierzu weiteres Potenzial ergeben.

Bei einer vom edacentrum [194] geführten Umfrage nach den Herausforderungen für künftige Chip-Designs ergaben sich unter anderem folgende Antworten [195]:

- Entwurfskomplexität und Systemintegration,
- Analog- und Mixed-Signal-Design,
- durchgehende Funktionale System-Verifikation,
- Zuverlässigkeit und Qualität,
- Vermeidung von bzw. Effektivitätssteigerung bei Re-Design,
- Herausforderungen im Bereich EDA für PCB-Design.

Das Problem der Entwurfskomplexität wird heute schon oftmals mit SystemC gelöst. Ebenso ist der durchgehende Entwurfs- und Validierungsablauf in einer einheitlichen

Umgebung bei der Erschaffung von SystemC verfolgt worden. Mit der hier präsentierten neuen Modellierung auf der Schalterebene in SystemC wird diesem Wunsch weiter entgegengekommen. Die Frage nach der Verlässigkeit kann mit dem Gebrauch der vorgestellten Techniken der Fehlerinjektion nachgewiesen und beantwortet werden. Ebenso ist das Vermeiden eines Re-Designs unter Zuhilfenahme der vorgestellten Co-Simulation lösbar. Der Punkt Analog- und Mixed-Signal-Design wurde in dieser Arbeit nicht behandelt, lässt sich aber zum Beispiel mit SystemC-AMS beantworten [48]. Hier kann man mit SystemC für verschiedene Aufgaben noch interessante Lösungen schaffen.

Bei all der Euphorie um SystemC wäre es jedoch zu viel verlangt, wenn alle Probleme des Entwurfes sich mit einer Sprache lösen ließen. Aufgaben, die z. B. im PCB-Bereich auftreten und somit überwiegend geometrische Probleme behandeln, wurden bisher noch nicht mit SystemC angegangen. Dieses Feld wird auch in nächster Zukunft sicher anderen EDA-Werkzeugen vorbehalten sein. Bestandteil eines hierarchischen Systementwurfs ist auch die Problematik der Syntheseschritte, also der Übergang von einer Entwurfsebene auf die nächste. In dieser Arbeit blieb diese Aufgabe außen vor, da sich die Prinzipien der Umsetzung bzw. Verfeinerung in SystemC nicht grundsätzlich von denen anderer HDLs unterscheiden.

Mit dieser Arbeit konnte gezeigt werden, dass die Simulation von Fehlern – insbesondere für die Fehlerinjektion und mit Beschleunigungsverfahren auch für die Fehlersimulation – in digitalen Schaltungen mit SystemC sinnvoll ist. Mit der Verwendung von SystemC besitzt man nicht nur für den Entwurf eine einzige Umgebung mit einem durchgängigen Design-Flow, sondern mit den vorgestellten Methoden gilt das nun auch für die Fehlerinjektion und Fehlersimulation. Der Umstieg von einer höheren Entwurfsebene, wo z. B. mit C++ Injektionen in SW stattfinden, zu den darunterliegenden, bei der eine HDL verwendet wird, kann jetzt für die Entwurfsumgebung und dem Beschreibungsmittel entfallen. Bei den vorgestellten Fehlerinjektionen lassen sich, im Vergleich zu VHDL, z. B. auch Low-Level-Modelle berechnen. Außerdem sind für die Fehlersimulation auch Modelle simulierbar, die sich aus unterschiedlichen Komponenten zusammensetzen. Es können somit heterogene Systeme untersucht werden, die für klassische Fehlersimulatoren ein Hindernis darstellen.



# Anhang

## A.1 Modelle und Beispiele

Im Folgenden sind diverse SystemC-Modelle aufgeführt. Die Darstellung erfolgt überwiegend und zweckmäßig als SystemC-Code.

### A.1.1 Beispiel für ein AND2-Gatter

---

```
1 // AND2 header file
2 #include "and2.h"

4 void and2::and2calc()
5 {
6     Z0.write (A0.read() & A1.read());
7 }
8
```

---

```
8 // AND2 header file
10 #include "systemc.h"

12 SC_MODULE (and2)
13 {
14     sc_out<sc_logic> z0;
15     sc_in<sc_logic> a0, a1;
16
17     void and2calc();
18
19     SC_CTOR (and2)
20     {
21         SC_METHOD (and2);
22         sensitive <<a0<<a1;
23     }
24 };
26
```

---

Listing A.1: 2-fach AND-Gatter in SystemC

## A.1.2 Einfacher PMOS-Transistor

---

```
// PMOS Switch
2 // 9-valued logic

4 #include "systemc.h"

6 SC_MODULE (pmos) {
    sc_out<sc_logic> z0;
8     sc_in<sc_logic> a0, ctrl;
    sc_logic a0_l, ctrl_l, res_l;
10     int a0_t, ctrl_t, res_t;

12     void doit();

14     SC_CTOR (pmos) {
        SC_METHOD (doit);
16         sensitive << a0 << ctrl;
    }
18 };

#include "pmos.h"
20
    const char pmosTab[4][4] = {
22 //      0      1      Z      X
        { '0', 'Z', 'L', 'L' },      //0
24         { '1', 'Z', 'H', 'H' },      //1
        { 'Z', 'Z', 'Z', 'Z' },      //Z
26         { 'X', 'Z', 'X', 'X' },      //X
    };
28
    void pmos::doit() {
30         a0_l = a0.read();
        a0_t = (int) a0_l.value() ;
32         ctrl_l = ctrl.read();
        ctrl_t = (int)ctrl_l.value();
34         res_l = pmosTab[a0_t] [ctrl_t];
        res_t = (int)res_l.value();
36         z0.write((sc_logic)pmosTab[a0_t] [ctrl_t]);
    }
}
```

---

Listing A.2: Einfacher PMOS-Transistor in SystemC

### A.1.3 Widerstandsbehafteter NMOS-Transistor

Das Listing A.3 zeigt einen NMOS-Transistor der Schalterebene mit Übertragungsverlust zwischen der Source-Drain-Strecke.

---

```

    // NMOS Switch - Resistive
2   // 9-valued logic

4   #include "systemc.h"

6   SC_MODULE (nmosr) {
    sc_out<sc_logic> z0;
8     sc_in<sc_logic> a0, ctrl;
    sc_logic a0_l, ctrl_l, res_l;
10    int a0_t, ctrl_t, res_t;

12    void doit();

14    SC_CTOR (nmosr) {
        SC_METHOD (doit);
16        sensitive << a0 << ctrl;
    }
18 };

```

---

```

#include "nmosr.h"
20
    const char nmosTab[9][9] = {
22 //      0      1      Z      X      L      H      U      W      D
    { 'Z', '0', 'L', 'L', 'Z', '0', 'X', 'L', 'X'}, //0
24    { 'Z', 'H', 'H', 'H', 'Z', 'H', 'W', 'H', 'W'}, //1
    { 'Z', 'W', 'X', 'X', 'Z', 'W', 'W', 'Z', 'W'}, //Z
26    { 'Z', 'X', 'X', 'X', 'Z', 'X', 'X', 'X', 'X'}, //X
    { 'Z', 'L', 'L', 'L', 'Z', 'L', 'W', 'L', 'W'}, //L
28    { 'Z', 'W', 'H', 'H', 'Z', 'W', 'W', 'H', 'W'}, //H
    { 'U', 'U', 'X', 'X', 'U', 'U', 'X', 'X', 'X'}, //U
30    { 'Z', 'W', 'W', 'W', 'Z', 'W', 'W', 'W', 'W'}, //W
    { 'Z', 'D', 'W', 'W', 'Z', 'D', 'W', 'W', 'W'}, //D
32 };

34 void nmosr::doit() {
    a0_l = a0.read();
36    a0_t = (int) a0_l.value() ;
    ctrl_l = ctrl.read();
38    ctrl_t = (int)ctrl_l.value();
    res_l = nmosTab[a0_t] [ctrl_t];
40    res_t = (int)res_l.value();
    z0.write((sc_logic)nmosTab[a0_t] [ctrl_t]);
42 }

```

---

Listing A.3: Widerstandsbehafteter NMOS-Transistor in SystemC

NMOS-R	ctrl								
data	0	1	Z	X	L	H	U	W	D
0	Z	0	L	L	Z	0	X	L	X
1	Z	H	H	H	Z	H	W	H	W
Z	Z	W	X	X	Z	W	W	Z	W
X	Z	X	X	X	X	X	X	X	X
L	Z	L	L	L	Z	L	W	L	W
H	Z	W	H	H	Z	W	W	H	W
U	U	U	X	X	U	U	X	X	X
W	Z	W	W	W	Z	W	W	W	W
D	Z	D	W	W	Z	D	W	W	W

Tabelle A.1: Auflösungstabelle für den verlustbehafteten NMOS-Transistor

#### A.1.4 Auflösungstabelle für 9-wertige Logik

```

const sc_logic_value_t
2 sc_logic_resolution_tbl[9][9] = {
  // 0 1 Z X L H U W DC
4  {Log_0, Log_X, Log_0, Log_X, Log_0, Log_0, Log_U, Log_0, Log_X}, //0
   {Log_X, Log_1, Log_1, Log_X, Log_1, Log_1, Log_U, Log_1, Log_X}, //1
6  {Log_0, Log_1, Log_Z, Log_X, Log_L, Log_H, Log_U, Log_W, Log_X}, //Z
   {Log_X, Log_X, Log_X, Log_X, Log_X, Log_X, Log_U, Log_X, Log_X}, //X
8  {Log_0, Log_1, Log_L, Log_X, Log_L, Log_X, Log_U, Log_W, Log_X}, //L
   {Log_0, Log_1, Log_H, Log_X, Log_X, Log_H, Log_U, Log_W, Log_X}, //H
10 {Log_U, Log_U, Log_U, Log_U, Log_U, Log_U, Log_U, Log_U, Log_U}, //U
    {Log_0, Log_1, Log_W, Log_X, Log_W, Log_W, Log_U, Log_W, Log_X}, //W
12 {Log_X, Log_X, Log_X, Log_X, Log_X, Log_X, Log_U, Log_X, Log_X} //DC
};

```

Listing A.4: Auflösungstabelle nach IEEE 1164 in SystemC

### A.1.5 Fredkin-Gatter mit Schalterebenenmodellierung

Im Folgenden ist das Fredkin-Gatter aus der reversiblen Logik wiedergegeben. Bei der Modellierung auf der Schalterebene besteht es überwiegend aus Transfer-Gattern, die wiederum aus Transistoren aufgebaut sind. Zusätzlich ist die Funktion in Form einer Wahrheitstabelle dargestellt.

X2	X1	X0	Y2	Y1	Y0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	0	1
1	1	1	1	1	1

Tabelle A.2: Symbol und Wahrheitstabelle für das Fredkin-Gatter

---

```

//FREDKIN.H: Fredkin-Gatter mit TG & Inverter
2
#ifdef FREDKIN_H
4 #define FREDKIN_H

6 #include "systemc.h"
#include "tg.h"
8 #include "inv.h"

10 SC_MODULE (fredkin) {
    sc_inout<sc_logic> x2y2;    //zusammengefasst
12     sc_inout_resolved x0,x1;
    sc_inout_resolved y0,y1;
14
    sc_signal<sc_logic> _x2y2;
16
    tg *tg1;
18     tg *tg2;
    tg *tg3;
20     tg *tg4;
    inv *inv1;
22
    SC_CTOR (fredkin) {
24         tg1 = new tg ("tg1");    //T-Gatter
        tg1->l0(x0);
26         tg1->r0(y0);
        tg1->ctrl(_x2y2);

```

---

---

```

28         tg1->_ctrl(x2y2);

30         tg2 = new tg ("tg2");
31         tg2->l0(x0);
32         tg2->r0(y1);
33         tg2->ctrl(x2y2);
34         tg2->_ctrl(_x2y2);

36         tg3 = new tg ("tg3");
37         tg3->l0(x1);
38         tg3->r0(y1);
39         tg3->ctrl(_x2y2);
40         tg3->_ctrl(x2y2);

42         tg4 = new tg ("tg4");
43         tg4->l0(x1);
44         tg4->r0(y0);
45         tg4->ctrl(x2y2);
46         tg4->_ctrl(_x2y2);

48         inv1 = new inv ("inv1");           //Inverter
49         inv1->a0(x2y2);
50         inv1->z0(_x2y2);
51     }
52 };
53 #endif

```

---

```

//TG.H: Transfer-Gatter
2
3 #ifndef TG_H
4 #define TG_H
5 #include "systemc.h"
6 #include "nmos.h"
7 #include "pmos.h"
8 #include "inv.h"

10 SC_MODULE (tg) {
11     sc_inout_resolved l0;
12     sc_inout_resolved r0;
13     sc_in<sc_logic> ctrl,_ctrl;
14
15     nmos *nmos1;
16     pmos *pmos1;

18     SC_CTOR (tg) {
19         nmos1 = new nmos ("nmos1");           //NMOS-Transistor
20         nmos1->a0(l0);
21         nmos1->z0(r0);
22         nmos1->ctrl(ctrl);

24         pmos1 = new pmos ("pmos1");           //PMOS-Transistor
25         pmos1->a0(r0);
26         pmos1->z0(l0);
27         pmos1->ctrl(_ctrl);

28         l0.initialize(sc_logic_Z);           //Start-Werte

```

---

---

```

30         r0.initialize(sc_logic_Z);
31     }
32 };
33 #endif

```

---

```

// NMOS.H: NMOS-Transistor - 9-wertige Logik
2
3 #ifndef NMOS_H
4 #define NMOS_H
5 #include "systemc.h"
6
7 SC_MODULE (nmos) {
8     sc_out<sc_logic> z0;
9     sc_in<sc_logic> a0, ctrl;
10    sc_logic a0_l, ctrl_l, res_l;
11
12    int a0_t, ctrl_t, res_t;
13
14    void doit();
15
16    SC_CTOR (nmos) {
17        SC_METHOD (doit);
18        sensitive << a0 << ctrl;
19        z0.initialize(sc_logic_Z);
20    }
21 };
22 #endif

```

---

```

//NMOS.CPP
23 #include "nmos.h"
24
25 const char nmosTab[4][4] = {
26     //      0      1      Z      X
27     { 'Z', '0', 'L', 'L' }, //0
28     { 'Z', '1', 'H', 'H' }, //1
29     { 'Z', 'Z', 'Z', 'Z' }, //Z
30     { 'Z', 'Z', 'X', 'X' }, //X, modifiziert: so keine Zyklen
31 };
32
33 void nmos::doit() {
34     a0_l = a0.read();
35     a0_t = (int) a0_l.value();
36     ctrl_l = ctrl.read();
37     ctrl_t = (int)ctrl_l.value();
38     z0.write((sc_logic)nmosTab[a0_t][ctrl_t]);
39 }
40

```

---

```

// PMOS.H: PMOS-Transistor, 9-wertige Logik
2
3 #ifndef PMOS_H
4 #define PMOS_H
5 #include "systemc.h"
6
7 SC_MODULE (pmos) {
8     sc_out<sc_logic> z0;
9     sc_in<sc_logic> a0, ctrl;

```

---

---

```

10     sc_logic a0_l, ctrl_l, res_l;

12     int a0_t, ctrl_t, res_t;

14     void doit();

16     SC_CTOR (pmos) {
17         SC_METHOD (doit);
18         sensitive << a0 << ctrl;
19         z0.initialize(sc_logic_Z);
20     }
21 };
22 #endif

```

---

```

//PMOS.CPP
2  #include "pmos.h"

4  const char pmosTab[4][4] = {
5      //      0      1      Z      X
6      { '0', 'Z', 'L', 'L' }, //0
7      { '1', 'Z', 'H', 'H' }, //1
8      { 'Z', 'Z', 'Z', 'Z' }, //Z
9      { 'Z', 'Z', 'X', 'X' }, //X, modifiziert: so keine Zyklen
10 };

12 void pmos::doit() {
13     a0_l = a0.read();
14     a0_t = (int) a0_l.value() ;
15     ctrl_l = ctrl.read();
16     ctrl_t = (int)ctrl_l.value();
17     z0.write((sc_logic)pmosTab[a0_t] [ctrl_t]);
18 }

```

---

Listing A.5: Fredkin-Gatter als Schaltermodell in SystemC



### A.1.6 Bidirektionaler Saboteur

---

```

1 // sab_bi.h
2 // Bidirektionaler Saboteur auf einer Leitung

4 #include "systemc.h"

6 SC_MODULE(saboteur_bi) {
7     sc_in<int> vf;                //Verzögerungsfaktor
8     sc_inout<sc_logic> in;        //Eingang/Ausgang 1-Bit
9     sc_in<int> richtung;          //Richtung=0: in->out =1: out->in
10    sc_in<int> fi;                //Ansteuerung der Fehlerinjektion
11    sc_inout<sc_logic> out;        //Ausgang/Eingang 1-Bit
12
13    bool v_d;
14
15    void stoere();
16
17    SC_CTOR(saboteur_bi) {
18        v_d = true;
19        SC_METHOD(stoere);
20        sensitive
21            << in
22            << out
23            << fi
24            << vf
25            << richtung;
26    }
27 };

```

---

```

28 // sab_bi.cpp
29 // Bidirektionaler Saboteur auf einer Leitung
30
31 #include "saboteur_bi.h"
32
33 void saboteur_bi::stoere() {
34     sc_time t_delay(1, SC_NS);    //Verzögerungsdauer =
35     t_delay *= vf.read();         // 1 ns * Faktor
36
37     int v_fi = fi.read();         //Auslesen der FI-Aktivierung
38
39     if (richtung) {               //Richtung = 1: out->in
40         switch (v_fi) {
41             case 0: in.write(out.read()); break;    //normal
42             case 1: in.write(sc_logic_0); break;    //sa0
43             case 2: in.write(sc_logic_1); break;    //sa1
44             case 3: in.write(~(out.read())); break; //bit-flip
45             case 4: in.write(sc_logic_X); break;    //'X'
46             case 5: in.write(sc_logic_Z); break;    //offen 'Z'
47             case 6: if (v_d) {                       //Verzög.
48                 next_trigger(t_delay);
49                 v_d=false;
50             }
51         }
52     } else {

```

---

---

```
52             in.write(out.read());
53             v_d=true;
54         }
55         break;
56     default: in.write(out.read()); break;    //normal
57 }
58 }
59 else {                                     //Richtung = 0: in->out
60     switch (v_fi) {
61         case 0: out.write(in.read()); break;    //normal
62         case 1: out.write(sc_logic_0); break;   //sa0
63         case 2: out.write(sc_logic_1); break;   //sa1
64         case 3: out.write(~(in.read())); break; //bitflip
65         case 4: out.write(sc_logic_X); break;   //'X'
66         case 5: out.write(sc_logic_Z); break;   //offen 'Z'
67             case 6: if (v_d) {                 //Verzög.
68                 next_trigger(t_delay);
69                 v_d=false;
70             }
71             else {
72                 out.write(in.read());
73                 v_d=true;
74             }
75             break;
76     default: out.write(in.read()); break;    //normal
77 }
78 }
```

---

Listing A.6: Universeller bidirektionaler Saboteur

### A.1.7 Variablen-Fehler in Mutanten: RAM-Modul

---

```

1  #include "systemc.h"
2  #include "shared.h"

4  SC_MODULE (mem) {
    sc_in<bool> en, rw, clk;
6    sc_in<sc_uint<ADDR_SIZE> > addr;
    sc_inout<sc_lv<WORD_SIZE> > data;
8    sc_in<sc_lv<WORD_SIZE> > fm;
    sc_in<int> fi;

10
    void doit();
12    sc_lv<WORD_SIZE> ram [MEM_SIZE];

14    SC_CTOR(mem) {
        SC_METHOD(doit);
16        sensitive_neg << clk;
            for (int i=0; I < MEM_SIZE; i++) ram[i]="00000000";
18    }
    };

```

---

```

20 #include "mem1.h"

22 void mem::doit() {
    sc_lv<WORD_SIZE> allZ (sc_logic_Z);
24    sc_lv<WORD_SIZE> allX (sc_logic_X);

26    if (en) {
        if (rw) { //read
28            if (addr.read() < MEM_SIZE)
                if (fi==1) {
30                    ram[addr.read()] |= fm.read();
                    data = ram[addr.read()];
32                }
                else data = ram[addr.read()];
34            else data = allX;
        }
36        else { //write
            if (addr.read() < MEM_SIZE)
38                if (fi==1) {
                    ram[addr.read()] = data.read();
40                    ram[addr.read()] |= fm.read();
                }
42                else ram[addr.read()] = data;
            }
44        }
        else data = allZ;
46    }

```

---

Listing A.7: Fehler in Variablen-Mutanten

### A.1.8 Parametrisierter Template-Mutant für ein AND-Gatter

---

```
1  #ifndef AND_H
2  #define AND_H
3  #include "systemc.h"
4
5  template <int size = 2>    //2-fach-AND als Voreinstellung
6  SC_MODULE(and) {
7
8      sc_out<sc_logic> z0;
9      sc_in<sc_logic> a[size];
10     sc_in<int> fi;
11
12     void doit();
13
14     SC_HAS_PROCESS (and);
15
16     and(sc_module_name name_) : sc_module(name_)
17     {
18         SC_METHOD(doit);
19         for (int i=0; i < size; i++) sensitive << a[i];
20     }
21 };
22
23 template <int size>
24 void and<size>::doit() {
25     int v-fi = fi.read();
26     sc_logic zt=a[0].read();
27     int i=1;
28     switch (v-fi) {
29         case 0: while (i<size) {                                //and[i]
30                 zt=zt & a[i].read();
31                 ++i;
32             }
33             z0.write(zt);
34             break;
35         case 1: z0.write(sc_logic_0); break;                    //sa0
36         case 2: z0.write(sc_logic_1); break;                    //sa1
37         default: while (i < size) {                               //and[i]
38                 zt = zt & a[i].read();
39                 ++i;
40             }
41             z0.write(zt);
42     }
43 }
44 #endif
```

---

Listing A.8: Template-AND-Mutant

### A.1.9 Parametrisierter Konstruktor-Mutant für ein AND-Gatter

---

```
1  #ifndef AND_H
2  #define AND_H
3  #include "systemc.h"
4
5  SC_MODULE(and) {
6      int size;
7      sc_out<sc_logic> z0;
8      sc_in<sc_logic> *a;
9      sc_in<int> fi;
10
11      void doit();
12
13      SC_HAS_PROCESS(and);
14      and(sc_module_name name_, int size_=2) : sc_module(name_) {
15          size = size_;
16          in = new sc_in<sc_logic>[size];
17          SC_METHOD(doit);
18          for (int i=0; i < size; i++) sensitive<<a[i];
19          sensitive << fi;
20      }
21  };
22 #endif
```

---

```
#include "and.h"
24
25 void and::doit() {
26     int v-fi = fi.read();
27     sc_logic zt = a[0].read(); //wir müssen draussen bleiben
28     int i=1;
29     switch (v-fi) {
30         case 0: while (i<size) {
31             zt=zt & a[i].read();
32             ++i;
33         }
34         z0.write(zt);
35         break; //and3
36         case 1: z0.write(sc_logic_0); break; //sa0
37         case 2: z0.write(sc_logic_1);
38             break; //sa1
39         default: while (i < size) {
40             zt=zt & a[i].read();
41             ++i;
42         }
43         z0.write(zt);
44     }
45 }
```

---

Listing A.9: Konstruktor-Argument-Mutant

### A.1.10 Auflösungstabellen für logischen Operationen der Simulatorkommandos

---

```

// AND-Operation
2  const sc_logic_value_t sc_logic::and_table[7][7] = {
    //      0      1      Z      X      B      A      R
4      { Log_0, Log_0, Log_0, Log_0, Log_0, Log_0, Log_0 }, //0
      { Log_0, Log_1, Log_X, Log_X, Log_0, Log_1, Log_X }, //1
8      { Log_0, Log_X, Log_X, Log_X, Log_0, Log_X, Log_X }, //Z
      { Log_0, Log_X, Log_X, Log_X, Log_0, Log_X, Log_X }, //X
10     { Log_0, Log_0, Log_0, Log_0, Log_0, Log_0, Log_0 }, //B
      { Log_0, Log_1, Log_X, Log_X, Log_0, Log_1, Log_X }, //A
12     { Log_0, Log_X, Log_X, Log_X, Log_0, Log_X, Log_R } //R
    };
14
// OR-Operation
16 const sc_logic_value_t sc_logic::or_table[7][7] = {
    //      0      1      Z      X      B      A      R
18     { Log_0, Log_1, Log_X, Log_X, Log_0, Log_1, Log_X }, //0
      { Log_1, Log_1, Log_1, Log_1, Log_1, Log_1, Log_1 }, //1
20     { Log_X, Log_1, Log_X, Log_X, Log_X, Log_1, Log_X }, //Z
      { Log_X, Log_1, Log_X, Log_X, Log_X, Log_1, Log_X }, //X
22     { Log_0, Log_1, Log_X, Log_X, Log_0, Log_1, Log_X }, //B
      { Log_1, Log_1, Log_1, Log_1, Log_1, Log_1, Log_1 }, //A
24     { Log_X, Log_1, Log_X, Log_X, Log_X, Log_1, Log_R } //R
    };
26
// XOR-Operation
28 const sc_logic_value_t sc_logic::xor_table[7][7] = {
    //      0      1      Z      X      B      A      R
30     { Log_0, Log_1, Log_X, Log_X, Log_0, Log_1, Log_X }, //0
      { Log_1, Log_0, Log_X, Log_X, Log_1, Log_0, Log_X }, //1
32     { Log_X, Log_X, Log_X, Log_X, Log_X, Log_X, Log_X }, //Z
      { Log_X, Log_X, Log_X, Log_X, Log_X, Log_X, Log_X }, //X
34     { Log_0, Log_1, Log_X, Log_X, Log_0, Log_1, Log_X }, //B
      { Log_1, Log_0, Log_X, Log_X, Log_1, Log_0, Log_X }, //A
36     { Log_X, Log_X, Log_X, Log_X, Log_X, Log_X, Log_R } //R
    };
38 // NOT-Operation
    const sc_logic_value_t sc_logic::not_table[7] =
40 { Log_1, Log_0, Log_X, Log_X, Log_1, Log_0, Log_R }; //~(01ZXBAR)

```

---

Listing A.10: Logische Operationen mit Simulatorkommandos

### A.1.11 Klasse MyInt für Beispiel von Simulationskommandos-Fehlerinjektionen in Variablen

---

```
2 // MyInt-Klasse zur Fehlerinjektion in Variablen
2 // Scheibzugriffe werden mit lock verboten

4 #ifndef MYINT_H
4 #define MYINT_H
6
6 #include "systemc.h"
8
8 class MyInt {
10     private:
10         int dat;
12         bool lock;

14     public:
14         // Constructor
16         MyInt (int dat_=0) {
16             this->dat = dat_;
18             lock = false;
18         }
20
20         // Destructor
22         ~MyInt()
22         {}
24
24         // Konvertierung - nur für dat
26         operator int() {
26             return this->dat;
28         }

30     inline bool operator == (const MyInt & rhs) const {
30         return (rhs.dat == dat && rhs.lock == lock);
32     }

34     inline MyInt& operator = (const MyInt& rhs) {
34         if (!lock) this->dat = rhs.dat;
36         else {cout <<"access locked"<<endl;} //event. Meldungen
36         return *this;
38     }

40     inline MyInt& operator = (int rhs) {
40         if (!lock) this->dat = rhs;
42         return *this;
44     }

44     // beide Variablen sind für das Debuggen interessant
46     inline friend void sc_trace(sc_trace_file *tf, const MyInt & v,
46         const sc_string& NAME ) {
48         sc_trace(tf,v.dat, NAME + ".dat");
48         sc_trace(tf,v.lock, NAME + ".lock");
50     }
```

---

```
52     inline friend ostream& operator << ( ostream& os,
        MyInt const & v ) {
54         os << v.dat;
        return os;
56     }

58 // Deklaration der Operanden
    inline friend MyInt operator + (const MyInt& v, const MyInt& w);
60 inline friend MyInt operator & (const MyInt& v, const MyInt& w);
    inline friend MyInt operator ~ (const MyInt& v);
62 inline friend MyInt operator / (const MyInt& v, const MyInt& w);
    inline friend MyInt operator * (const MyInt& v, const MyInt& w);
64 inline friend MyInt operator | (const MyInt& v, const MyInt& w);
    inline friend MyInt operator - (const MyInt& v, const MyInt& w);
66 inline friend MyInt operator ^ (const MyInt& v, const MyInt& w);

68 public:
    inline MyInt operator &= ( const MyInt& b ) {
70         if (!lock) this->dat = dat & b.dat;
        return this->dat;
72     }

74     MyInt operator ++ () // prefix
    {
76         ++dat;
        return this->dat;
78     }

80     MyInt operator ++ (int) // postfix
    {
82         int aa = this->dat;
        dat++;
84         return aa;
    }
86
    // lock+unlock+getlock
88     inline bool setlock() {
        this->lock=true;
90         return true;
    }
92
    inline bool unsetlock() {
94         this->lock=false;
        return false;
96     }

98     inline bool getlock () {
        return this->lock;
    }
100 };

102 // Überladen der Operatoren - keine extra-lock-Abfrage nötig -
    // diese ist schon durch die Zuweisung implementiert
104
```

---



---

```
    MyInt operator+ (const MyInt& v, const MyInt& w ) {
106         MyInt summe;
            summe.dat = v.dat + w.dat;
108         return summe;
    }
110
    MyInt operator& (const MyInt& v, const MyInt& w ) {
112         MyInt result;
            result.dat = v.dat & w.dat;
114         return result;
    }
116
    MyInt operator~ (const MyInt& v) {
118         if (!(v.lock)){
            MyInt result;
120             result.dat = ~v.dat;
            return result;
122         }
    }
124
    MyInt operator/ (const MyInt& v, const MyInt& w ) {
126         MyInt result;
            if (w.dat==0)
128                 result.dat=0;
            else
130                 result.dat=((int) (v.dat/w.dat)) ;
            return result;
132 }

134 MyInt operator* (const MyInt& v, const MyInt& w ) {
            MyInt result;
136             result.dat = v.dat * w.dat;
            return result;
138 }

140 MyInt operator| (const MyInt& v, const MyInt& w ) {
            MyInt result;
142             result.dat = v.dat | w.dat;
            return result;
144 }

146 MyInt operator- (const MyInt& v, const MyInt& w ) {
            MyInt result;
148             result.dat = v.dat - w.dat;
            return result;
150 }

152 MyInt operator^ (const MyInt& v, const MyInt& w ) {
            MyInt result;
154             result.dat = v.dat ^ w.dat;
            return result;
156 }

158 #endif
```

---

Listing A.11: Beispielklasse MyInt für Fehlerinjektion in Variablen

## A.2 Code-Mutationen in verhaltensorientierten HDL-Modellen

Fehlerklassen sind für verhaltensorientierte Modelle in Form von VHDL-Beschreibungen schon seit dem Beginn der 90er Jahre bekannt und z. B. in [152] beschrieben. Im Folgenden ist eine mögliche Systematik, eingeteilt in acht Fehlerklassen, aufgeführt. Die hier vorgestellten Beispiele wurden, auch zum besseren Verständnis, in eine entsprechende SystemC-Beschreibung überführt. Die Fehlerinjektionen werden durch syntaktische Änderungen erreicht. Bei den Beispielen wird zunächst der fehlerfreie Original-Code und dann der modifizierte fehlerhafte Code vorgestellt.

### 1.) Stuck-Then

Der *stuck-then*-Fehler tritt in einer *if-then-else*-Bedingung auf und führt dazu, dass der *else*-Zweig immer ausgeschlossen ist.

---

```
if (logischer Ausdruck)
    z.write(sc_logic_1);
else
    z.write(sc_logic_0);
```

---

---

```
if (true)
    z.write(sc_logic_1);
else
    z.write(sc_logic_0);
```

---

In dem Beispiel wird mit der Ersetzung des logischen Ausdrucks durch den statischen *true*-Wert erreicht, so dass an den Port *z* immer das Schreiben einer logischen '1' erfolgt.

### 2.) Stuck-Else

Der *stuck-else*-Fehler tritt in einer *if-then-else*-Bedingung auf und führt dazu, dass der *then*-Zweig immer ausgeschlossen ist. Er stellt das Gegenstück zum *stuck-then*-Fehler dar.

---

```
if (logischer Ausdruck)
    z.write(sc_logic_1);
else
    z.write(sc_logic_0);
```

---

---

```
if (false)
    z.write(sc_logic_1);
else
    z.write(sc_logic_0);
```

---

In dem Beispiel wird mit der Ersetzung des logischen Ausdrucks durch den statischen *false*-Wert erreicht, so dass an den Port *z* immer das Schreiben einer logischen '0' erfolgt.

### 3.) Assignment-Control

Der *assignment-control*-Fehler bezeichnet einen Irrtum des Zuweisungsoperators bei der Zuweisung eines neuen Wertes an ein Signal.

---

```
A Sig = neuer Wert;
```

---

---

```
A Sig = A Sig;
```

---

Der Fehlereinbau wird so realisiert, dass eine Ersetzung der Zuweisung des neuen Wertes durch die eigene Bezeichnung erfolgt. Für Ports, speziell für die Ausgänge, wird Ähnliches vorgeschlagen. Bei diesen darf der Zuweisungsbefehl auskommentiert werden.

### 4.) Dead-Process

Der *dead-process*-Fehler ist ein Ausfall der Ausführung, der die Anweisungen innerhalb eines Prozesses betrifft.

---

```
SC_METHOD (doit);  
    sensitive << a << b << c;
```

---

---

```
SC_METHOD (doit);  
    sensitive << Sig_Internal_Static;  
//    sensitive << a << b << c;
```

---

Ein *dead-process* wird durch eine Änderung der Sensitivitätsliste erreicht. Im Beispiel wird der Prozess *doit* durch eine Änderung bei den Signalen *a*, *b*, *c* aufgerufen. Die Prozess-Sensitivierung kann durch ein statische Signal, welches sich im Programmverlauf nie ändert oder noch einfacher durch eine Auskommentierung, deaktiviert werden.

### 5.) Dead-Clause

Der *dead-clause*-Fehler bewirkt eine Veränderung in einem *switch-case*-Konstrukt. Für mindestens eine *case*-Marke wird der Fehlerfall implementiert.

---

```
switch (var_wert) {  
    case 0:    Sig_1 = sc_logic_X;  
              Sig_2 = sc_logic_X;  
              break;  
    case 1:    Sig_1 = sc_logic_1;  
              Sig_2 = sc_logic_1;  
              break;  
    . . .
```

---

---

```
switch (var_wert) {  
    case 0:      Sig_1 = sc_logic_X;  
                Sig_2 = sc_logic_X;  
                break;  
    case 1:      Sig_1 = Sig_1;  
                Sig_2 = Sig_2;  
                break;  
    . . .  
}
```

---

Im Beispiel ist der *dead-clause* so realisiert, dass für die zweite *case*-Marke eine Signalzuweisung auf sich selbst passiert. Eine Wertänderung bleibt für diesen Anweisungsblock ausgeschlossen.

### 6.) Micro-Operation

Ein *micro-operation*-Fehler meint einen Ausfall eines Operators bezüglich seiner beabsichtigten Operation. Typischerweise wird ein Operator in einem Befehl durch einen anderen Operator aus der gleichen Klasse ausgetauscht.

---

```
Out.write( In1.read() & In2.read() );
```

---

```
Out.write( ~(In1.read() & In2.read()) );
```

---

Im Beispiel wird die AND-Verknüpfung durch eine NAND-Operation ersetzt.

### 7.) Local-Stuck-Data

Der *local-stuck-data*-Fehler bezeichnet die fehlerhafte Änderung einer Variablen oder eines Signals in einem lokalen Ausdruck. Weitere Zuweisungen im Programmcode bleiben unberührt.

---

```
if (a.read() == 25) out.write(50);  
. . .  
if (a.read() == 2) out.write(4);
```

---

```
if (a.read() == 25) out.write(50);  
. . .  
if (a.read() == 2) out.write(0);    //Fehlerfall
```

---

Für das Beispiel wird im Fehlerfall keine 4 sondern eine 0 in den Ausgang *out* geschrieben. Der Wert 0 stellt den Fehler dar.

### 8.) Global-Stuck-Data

Der *global-stuck-data*-Fehler bezeichnet die fehlerhafte Änderung einer Variablen oder eines Signals in einer Architektur für mehrere oder alle entsprechenden Ausdrücke.

---

```
if (a.read() == 25) out.write(50);  
.  
.  
if (a.read() == 2) out.write(4);
```

---

```
if (a.read() == 25) out.write(0);    //Fehlerfall  
.  
.  
if (a.read() == 2) out.write(0);    //Fehlerfall
```

---

Für das Beispiel werden alle Zuweisungen mit dem fehlerhaften Wert beschrieben.

## A.3 Überblick zu den Simulationsmodellen

Im Folgenden sind einige Eigenschaften zu den verwendeten Modellen der verschiedenen Beispielsimulationen aufgeführt.

Bezeichnung	Art/Typ	Charakteristiken
c17	kombinatorisch	6 Gatter
c17(sl)	Schalterebene	24 Transistoren + 6 Buffer
c432	27-Kanal-Interrupt-Controller	36 Eingänge, 7 Ausgänge, 160 Gatter
c432(sl)	27-Kanal-Interrupt-Controller	906 Transistoren + 214 Buffer
s35932*	sequenziell	35 Eingänge, 16065 Gatter, 320 Ausgänge, (1728 FFs)
add8	Funktionales RTL-Modell eines 8-Bit-Addierers	-
alup8	Funktionales RTL-Modell einer 8-Bit-Parallel-Recheneinheit	-
MSAB	kombinatorisch	8 Eingänge, 5 Ausgänge, 35 Gatter
MSAB(sl)	Schalterebene	178 Transistoren + 35 Buffer
TGB1(sl)	Schalterebene	12 Transistoren
TGB2(sl)	Schalterebene	36 Transistoren
MUX16(sl)	T-Gatter + Inverter	90 Transistoren
TGB1	Gatterebene	6 T-Gatter
TGB2	Gatterebene	18 T-Gatter
MUX16	Gatterebene	30 Gatter
a16_x_behave	Funktionales RTL-Modell eines 16-Bit-Addierers	-
a16_x_gate	Gatterebene	80 Gatter
fsm6	Gatterebene	17 Gatter
decod	Gatterebene	31 Gatter, Decoder

\* *modifiziert für Scan Path*

Tabelle A.3: Beispielschaltungen in SystemC

## A.4 MPS-Klassendiagramm

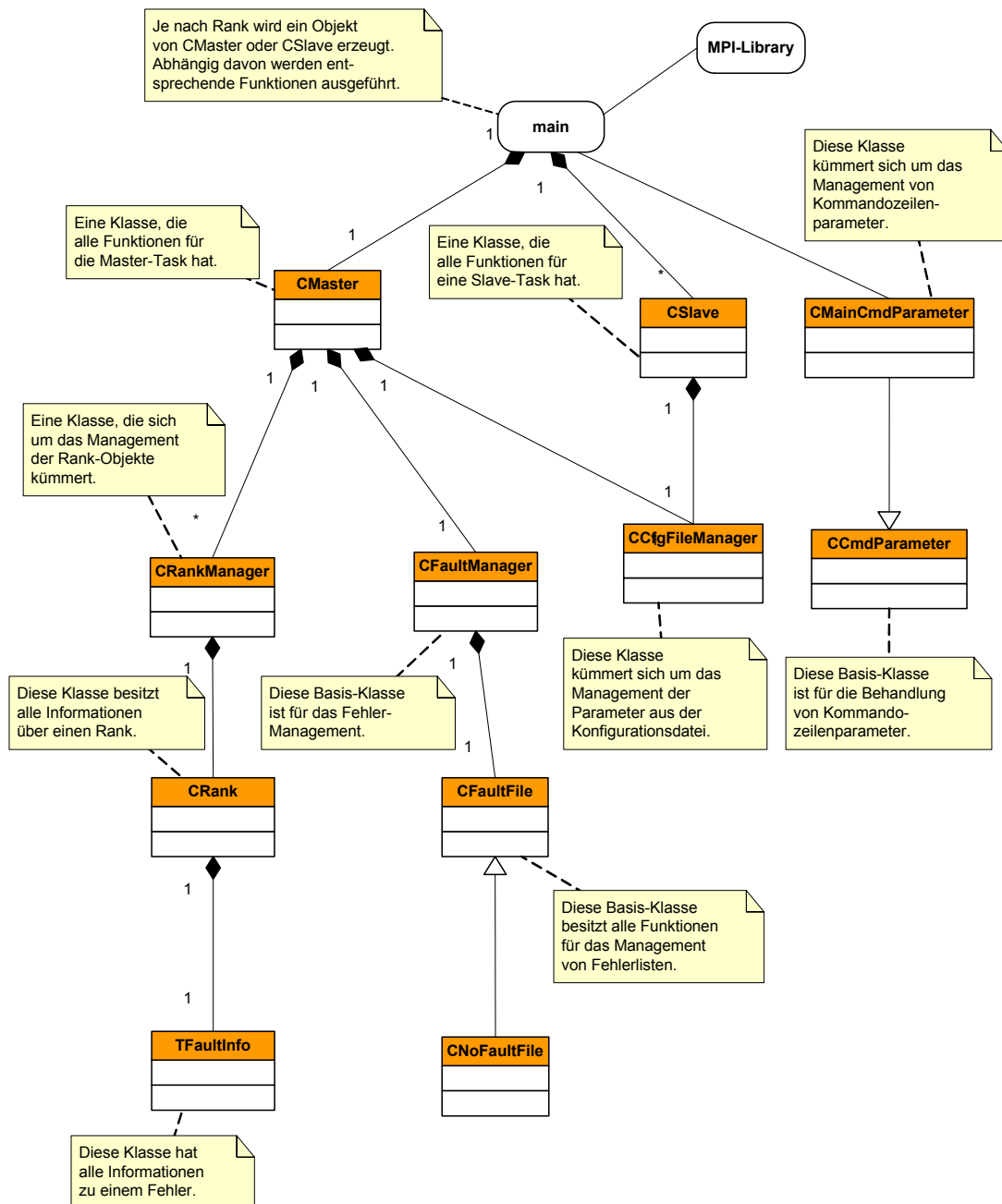
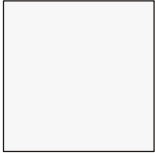






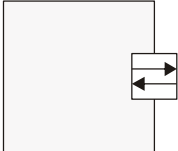


Abbildung A.1: MPS-Klassendiagramm

## A.5 grafische SystemC-Darstellungen

Symbol	Bezeichnung
	Modul
	Process
	Port
	Interface
	Primitive Channel
	Directed Primitive Channel
	Channel
	Modul mit Port



# Abkürzungsverzeichnis

ALU	Arithmetic Logical Unit
ATPG	Automatic Test Pattern Generation
CFI	Clock for Fault Injection
CLK	Clock
CMOS	Complementary Metal Oxide Semiconductor
CPL	Complementary Pass Transistor Logic
CPU	Central Processing Unit
DLL	Dynamic Link Library
DPL	Double Pass Transistor Logic
DUT	Device Under Test oder Design Under Test
DUV	Device Under Verification
EDA	Electronic Design Automation
FI	Fault Injection
FIT	Fault Injection Tool
FLI	Foreign Language Interface
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
FÜ	Fehlerüberdeckungsgrad
Gnd	Ground
GUI	Graphical User Interface
HDL	Hardware Description Language
HLN	High-Level-Description
HW	Hardware
HWFI	Hardware Fault Injection
IDDQ	Current Drain Direct Quite
IM	Injektionsmanager
IP	Intellectual Property
KLT	Kernel-Level Threads
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MoC	Model of Computation
MOS	Metal Oxide Semiconductor
MPI	Message Passing Interface
MPS	Multi-Processing Simulation

NMOS	N-type Metal-Oxide Semiconductor
OOP	Objektorientierte Programmierung
OSCI	Open SystemC Initiative
OSI	Open Source Initiative
PC	Personal Computer
PCB	Printed Circuit Board
PDDT	Parallel-definierter Datentyp
PLA	Programmable Logic Array
PLI	Programmable Logic Interface
PMOS	P-type Metal-Oxide Semiconductor
PPSF	parallel pattern single fault simulation
SPICE	Simulation Program with Integrated Circuit Emphasis
PTL	Pass Transistor Logic
PVM	Parallel Virtual Machine
RTL	Register Transfer Level
sa0	stuck-at-0
sa1	stuck-at-1
SCV	SystemC Verification Library
SEU	Single Event Upset
SFI	Simulated Fault Injection
SIMD	Single Instruction Multiple Data
SIP	Systems in Package
SISD	Single Instruction Single Data
SPPF	single pattern parallel fault simulation
SPSF	single pattern single fault simulation
SoC	System on a Chip
SW	Software
SWFI	Software Fault Injection
TBL	Table File
TPG	Test Pattern Generation
ULT	User-Level Threads
UML	Universal Modeling Language
UNIX	Uniplexed Information and Computing System
Vdd	Voltage (multiple drains)
VERILOG	eine Hardware-Beschreibungssprache
VHDL	VHSIC Hardware Description Language
VHPI	VHDL Procedural Interface
VLWI	Very Long Instruction Word

# Symbolverzeichnis

	logisch OR
&	logisch UND
^	logisch XOR
~	Eins-Komplement
$A_b$	Anzahl der verwendeten Bits
$A_{SS}$	Anzahl aller Simulationsschritte
$A_{SSF}$	Anzahl der Simulationsschritte für Fehlerinjektionen
$A_{SSG}$	Anzahl der Simulationsschritte für Gut-Simulation
$C_{par}$	Kosten bei paralleler Ausführung
$E_{par}$	Effizienz bei paralleler Ausführung
$F$	Fehleranzahl
$G$	Testmuster für Gutsimulation
$f$	serieller Bruchteil
$F_M$	Fehlermodell
$F\ddot{U}$	Fehlerüberdeckung
$KN$	Anzahl von Knoten
$K$	Kommunikationsaufwand
$L$	Testsatzlänge
$M$	Anzahl von Komponenten
$N$	Anzahl von Gattern
$P$	Anzahl von Prozessoren
$S$	Anzahl von Simulationen
$S_P$	Speed-up
$T_{FS}$	Fehlersimulationszeit
$T_G$	Rechenzeit pro Gatter
$T_{par}$	Ausführungszeit in paralleler Form
$T_S$	Simulationszeit
$T_{ser}$	Ausführungszeit in serieller Form
$t_{ser}$	im seriellen Teil verbrachte Zeit
$T_{FSL}$	Fehlersimulationszeit mit Berücksichtigung der Testsatzlänge



# Abbildungsverzeichnis

Abbildung 2.1: Y-Diagramm nach Gajski/Walker [191] .....	11
Abbildung 2.2: Architektur der SystemC-Klassenbibliothek.....	13
Abbildung 2.3: Vergleich zwischen herkömmlichen HDL- und SystemC-Entwurf.....	14
Abbildung 2.4: Ebenen beim durchgängigen Entwurf .....	15
Abbildung 2.5: SystemC-Projektumgebung.....	19
Abbildung 3.1: Übersicht von Arten der Logiksimulation.....	22
Abbildung 3.2: Simulationszyklen bei der ereignisgesteuerten Simulation.....	25
Abbildung 3.3: Simulationszyklen bei der zyklenbasierten Simulation .....	25
Abbildung 3.4: Ablauf einer einfachen Fehlersimulation .....	28
Abbildung 3.5: Rechenaufwand bei einer kombinatorischen Schaltung.....	28
Abbildung 3.6: Musterparallele Berechnung in einem 4-Bit-Maschinenwort .....	32
Abbildung 3.7: Fehlerparallele Berechnung in einem 4-Bit-Maschinenwort .....	33
Abbildung 4.1: Zusammenhang zwischen Schaltungstechnologien, Verhalten und Fehlermodellen .....	38
Abbildung 4.2: FSM einer einfachen Temperaturreglersteuerung .....	39
Abbildung 4.3: AND-Modul mit 5 ns Verzögerung .....	41
Abbildung 4.4: <i>NMOS-Primitive</i> der Schalterebene .....	45
Abbildung 4.5: a) AOI32-Komplexgatter mit b) Logikgatterauflösung .....	48
Abbildung 4.6: Struktur des AOI32-Gatters in SystemC .....	50
Abbildung 4.7: a) Schaltung und b) Symbol eines CPL-AND/NAND-Gatters.....	51
Abbildung 4.8: Pseudo-Code der Unterbrechungsfehlermodellierung im CPL-Gatter ..	53
Abbildung 4.9: Widerstandsverhalten von NMOS- und PMOS-Transistor.....	53
Abbildung 4.10: a) Zusammenschaltung und b) Symbol des Transfer-Gatters .....	54
Abbildung 4.11: a) Bus-Anschaltung eines Transfer-Gatters und b) Schalterverhalten	54
Abbildung 4.12: Fehler im Lebenszyklus eines Systems.....	57
Abbildung 4.13: Eine Klassifizierung von Fehlern .....	58
Abbildung 4.14: Geometrische Fehler durch zu viel und zu wenig Material.....	60
Abbildung 4.15: Haftfehlermodellierung a) Gatterebene und b) Transistorebene.....	62
Abbildung 4.16: SystemC-Modul für Haftfehlermodellierung .....	64
Abbildung 4.17: Stuck-at-1-Fehler mit OR-Gatter: a) Originalmodell, b) modifiziert..	65
Abbildung 4.18: Wired-OR-Brückenfehler .....	66
Abbildung 4.19: AND-Gatter mit a) realem und b) logischem Verzögerungsverhalten	71
Abbildung 4.20: Mehrfachтакты für kurze transiente Fehler .....	74
Abbildung 4.21: SystemC-Modell für kurze transiente Fehler .....	75
Abbildung 5.1: Zeitlicher Zusammenhang zwischen Fault, Error, Failure .....	80

Abbildung 5.2: Fehlerraum .....	80
Abbildung 5.3: FARM-Mengen .....	81
Abbildung 5.4: Eine Systematik von simulierten Fehlerinjektionen bei HDLs .....	85
Abbildung 5.5: AMLETO-Architektur .....	91
Abbildung 5.6: Architektur der Fehlerinjektion .....	92
Abbildung 5.7: Fehlerinjektion in das funktionale Modell .....	94
Abbildung 5.8: Prinzipielle Wirkungsweise eines Saboteurs .....	96
Abbildung 5.9: Einfacher serieller Saboteur .....	96
Abbildung 5.10: Komplexer serieller Saboteur .....	98
Abbildung 5.11: Paralleler Saboteur .....	101
Abbildung 5.12: Pseudo-Code für DUT-Saboteur-Modifikation .....	102
Abbildung 5.13: Testbench-Dateien nach der Simulationsvorbereitung .....	103
Abbildung 5.14: AND/NAND-Mutant durch Austauschen .....	105
Abbildung 5.15: Erweiterter RAM-Mutant .....	106
Abbildung 5.16: Pseudo-Code für DUT-Mutant-Modifikation .....	114
Abbildung 5.17: Allgemeine Injektion mit Simulationskommandos .....	115
Abbildung 6.1: FIT-Projekt .....	130
Abbildung 6.2: Mixed-Simulations-Projekt .....	134
Abbildung 6.3: Vergleich verschiedener Mixed-Language/Co-Simulationen .....	139
Abbildung 7.1: Einfaches Schichtenmodell zum Parallelisierungsgrad .....	143
Abbildung 7.2: Aufbau des PC-Clusters .....	161
Abbildung 7.3: Schichtenbasiertes Software-Modell .....	162
Abbildung 7.4: Ablauf für Simulationen mit mehreren Einzelfehlern .....	163
Abbildung 7.5: Programmablaufplan für den Master-Prozess .....	165
Abbildung 7.6: Programmablaufplan für den Slave-Prozess .....	166
Abbildung 7.7: Schema der Simulationsprojekte für die verteilte Simulation .....	168
Abbildung 7.8: Speed-up für einige ISCAS-Schaltungen .....	169
Abbildung A.1: MPS-Klassendiagramm .....	199

# Tabellenverzeichnis

Tabelle 4.1: Erweitertes <i>NMOS-Primitive</i> der Schalterebene .....	46
Tabelle 4.2: Fehlerverhalten des CPL-AND/NAND-Gatters mit Unterbrechung .....	52
Tabelle 4.3: Simulationszeiten von Modellen der Gatter- und Schalterebene .....	57
Tabelle 4.4: Abstraktionsebene und Fehlermodelle .....	61
Tabelle 4.5: Funktionstabelle für NAND und Fehlermodelle .....	63
Tabelle 5.1: Simulationsergebnisse für (nicht-)parametrisierte Beispiele.....	114
Tabelle 5.2: Neue Auflösungstabelle in der Klasse <i>sc_signal_resolved</i> .....	118
Tabelle 5.3: AND-Funktionstabelle für Simulationskommandos .....	119
Tabelle 5.4: Simulationsergebnisse für verschiedene Injektionstechniken .....	124
Tabelle 6.1: Simulationen der Mixed-Language-Co-Simulation .....	138
Tabelle 7.1: Simulationsergebnisse für Bit-parallele Beispiele .....	152
Tabelle 7.2: Operationsauflösung des selbstdefinierten Logikvektors.....	154
Tabelle 7.3: Simulationsergebnisse für Vektor-parallele Beispiele .....	155
Tabelle 7.4: Simulationsergebnisse für Array-parallele Beispiele .....	157
Tabelle 7.5: Speed-up-Simulationsergebnisse für Beispiele höherer Ebene.....	159
Tabelle 7.6: Kombination paralleler Verfahren.....	171
Tabelle A.1: Auflösungstabelle für den verlustbehafteten NMOS-Transistor .....	180
Tabelle A.2: Symbol und Wahrheitstabelle für das Fredkin-Gatter .....	181
Tabelle A.3: Beispielschaltungen in SystemC .....	198





# Listings

Listing 4.1: FSM einer Temperatursteuerung in SystemC .....	40
Listing 4.2: AND-Modul mit Verzögerung in SystemC .....	41
Listing 4.3: Einfacher NMOS-Transistor in SystemC nach IEEE 1364 .....	45
Listing 4.4: AND-Modul mit Verzögerung in SystemC .....	49
Listing 4.5: Transfer-Gatter mit Fehlersignalisierung .....	55
Listing 4.6: Transfer-Gatter mit Transistoren der Schalterebene .....	56
Listing 4.7: Haftfehlermodellierung in SystemC .....	64
Listing 4.8: Haftfehlermodellierung in SystemC mit arithmetischer Operation .....	65
Listing 4.9: Wired-Fehler-Modul in SystemC .....	67
Listing 4.10: Unbestimmt-Fehler-Modul in SystemC .....	68
Listing 4.11: Erweiterung des NMOS-Transistors mit stuck-on-Fehler .....	69
Listing 4.12: Einfacher stuck-open-Fehler in SystemC .....	70
Listing 4.13: Einfacher Verzögerungsfehler für ein AND-Gatter .....	71
Listing 4.14: Detailliertes Verzögerungsmodell des AND-Gatters .....	72
Listing 4.15: Bit-Flip-Fehlermodell in SystemC .....	74
Listing 5.1: SystemC-Code für einen seriellen Saboteur .....	98
Listing 5.2: SystemC-Code für komplexen seriellen Saboteur mit Signaldominanz .....	99
Listing 5.3: SystemC-Code für einen komplexen seriellen Saboteur .....	100
Listing 5.4: Paralleler Saboteur - Instanziierung .....	101
Listing 5.5: Mutant in SystemC .....	107
Listing 5.6: Mutant in SystemC für Variablen-Manipulation in einem RAM-Modul .....	109
Listing 5.7: Template-Mutant mit AND-Funktion .....	111
Listing 5.8: Instanziierung des Template-Mutanten .....	111
Listing 5.9: Konstruktor-Mutant mit AND-Funktion .....	112
Listing 5.10: Instanziierung des Konstruktor-Mutanten .....	113
Listing 5.11: Simulations- und Fehlerinjektionssteuerung in der main.cpp .....	119
Listing 5.12: Selbstdefinierter Datentyp zur Fehlerinjektion .....	121
Listing 5.13: Saboteur mit Simulations-Kommando .....	123
Listing 6.1: Änderungen in der sc_main.cpp .....	133
Listing 6.2: calculate_1-Routine im Interface-Teil .....	136
Listing 6.3: Testbench im Interface-Teil .....	137
Listing 7.1: Schneller fehlerparalleler Saboteur in SystemC .....	151
Listing 7.2: NOR-Funktion mit selbstdefiniertem Vektor .....	154
Listing 7.3: Parallel-definierter Datentyp .....	158
Listing A.1: 2-fach AND-Gatter in SystemC .....	177

Listing A.2: Einfacher PMOS-Transistor in SystemC .....	178
Listing A.3: Widerstandsbehafteter NMOS-Transistor in SystemC .....	179
Listing A.4: Auflösungstabelle nach IEEE 1164 in SystemC .....	180
Listing A.5: Fredkin-Gatter als Schaltermodell in SystemC .....	184
Listing A.6: Universeller bidirektionaler Saboteur .....	186
Listing A.7: Fehler in Variablen-Mutanten .....	187
Listing A.8: Template-AND-Mutant .....	188
Listing A.9: Konstruktor-Argument-Mutant .....	189
Listing A.10: Logische Operationen mit Simulatorkommandos .....	190
Listing A.11: Beispielklasse MyInt für Fehlerinjektion in Variablen .....	193

# Literaturverzeichnis

- [1] Peter Marwedel: Synthese und Simulation von VLSI-Systemen, Carl Hanser Verlag München 1993, ISBN 3-446-16146-5
- [2] Peter Marwedel: Eingebettete Systeme, Springer-Verlag 2007, ISBN: 978-3-540-34048-5
- [3] Göran Herrmann, Dietmar Müller: ASIC – Entwurf und Test, Fachbuchverlag Leipzig 2004, ISBN 3-446-21709-6
- [4] Marco Beyer: Eine FPGA/DSP-Entwicklungsplattform für eingebettete audiosignalverarbeitende Echtzeitsysteme, Dissertation Berlin 2003
- [5] Hans-Joachim Wunderlich: Hochintegrierte Schaltungen: Prüfunggerechter Entwurf und Test, Springer Verlag Berlin 1991, ISBN 3-540-53456-3
- [6] P. Goel: Test Generation Costs Analysis and Projections; Proceedings 17th. Design Automation Conference, pp.77-84, June 1980
- [7] Wilfried Daehn: Testverfahren in der Mikroelektronik. Methoden und Werkzeuge, Springer Verlag Berlin 1997, ISBN 3-540-61728-0
- [8] Hans Wojtkowiak: Test und Testbarkeit digitaler Schaltungen, Teubner Verlag Stuttgart 1988, ISBN 3-519-02263-X
- [9] Schulz, M. H.; Trischler E.; Sarfert, T.: Socrates: A Highly Efficient Automatic Test Pattern Generation System, IEEE Trans. Computer-Aided Design. 7 (1), 1988 S.126-137
- [10] John Willis, et al: MinSim: Optimized, Compiled VHDL Simulation Using Networked & Parallel Computers, IEEE VHDL International User Forum, October 1993, pp. 137-144
- [11] Gläser, U. Hübner, U. Vierhaus, H.T. : Mixed Level Hierarchical Test Generation for Transition Faults and Overcurrent Related Defects, Test Conference 1992, Proceedings, ISBN 0-7803-0760-7
- [12] Jürgen Alt: Fehlersimulation synchroner Schaltungen unter Berücksichtigung nicht-klassischer Fehler, Dissertation, Fakultät für Maschinenwesen Universität Hannover 1995.

- [13] Michael H. Schulz: Testmustergenerierung und Fehlersimulation in digitalen Schaltungen mit hoher Komplexität, Springer Verlag Berlin 1988, ISBN 3-540-50051-0
- [14] Alfred Kölbl: Verifikation digitaler Schaltungen mittels symbolischer Simulation, Dissertation, Shaker Verlag Aachen 2003, ISBN 3-8322-1159-4
- [15] Ozgur Sinanoglu, Alex Orailoglu: RT-level Fault Simulation Based on Symbolic Propagation, 19th IEEE VLSI Test Symposium, March 2001, pp. 0240
- [16] D. B. Armstrong: A Deductive Method for Simulating Faults in Logic Circuits, IEEE Transactions on Computers 1972, C-21, No. 5, pp. 464-471
- [17] E.G. Ulrich, T. Baker: The Concurrent Simulation of Nearly Identical Digital Networks, Proc. 1973 Design Automation Workshop, pp. 145-150
- [18] Ulrich, E.: Concurrent Simulation at the Switch, Gate and Register Levels, ITC 1985, pp.703-709
- [19] Nikolaus Goders, Reinard Kaibel: PARIS: A Parallel Pattern Fault Simulator for Synchronous Sequential Circuits, In Proc. Int. Conf. on Computer-Aided Design, pp. 542-545, 1991.
- [20] Kung, Lin: Parallel sequence fault simulation for synchronous sequential circuits, Journal of Electronic Testing, 1996 pp. 267-277.
- [21] H. K. Lee and D. S. Ha, „HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits“, Proc. ACM-IEEE Design Automation Conference (DAC) 1992, pp. 336-340.
- [22] Thomas M. Neirmann, Wu-Tung Cheng and Janak H. Patel, PROOFS: A fast, memory-efficient sequential circuit fault simulator, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Feb. 1992, Volume: 11 , Issue: 2.
- [23] Alexander Miczo: Digital Logic Test and Simulation, Wiley & Sons 2nd edition Hoboken, New Jersey, Juli 2003, ISBN 0-471-43995-9
- [24] Mentor Graphics Inc.: Modelsim, [www.modelsim.com](http://www.modelsim.com), 05.02.2007
- [25] Aldec Inc.: Active-HDL und Riviera, [www.aldec.com](http://www.aldec.com), 1.11.2005.
- [26] Aldec Inc.: Riviera – SystemC Primer 1.1, <http://www.aldec.com/Registration/Download/Default.aspx?p=SystemC-Primer&v=1.1&platform=Windows>, 21.04.2007
- [27] Dolphin Integration: SMASH, <http://www.dolphin.fr/medal/smash>, 05.02.2007
- [28] Cadence Design Systems, Inc.: Standards and Languages, <http://www.cadence.com/partners/languages/languages.aspx>, 25.04.2007

- [29] SystemC Community: [www.systemc.org](http://www.systemc.org), 01.01.2007.
- [30] IEEE Package 1164: <http://ieeexplore.ieee.org/xpl/standards.jsp>, 29.01.2007
- [31] Hiren D. Patel, Sandeep K. Shukla: Towards a heterogeneous simulation kernel for system level models: a SystemC kernel for synchronous data flow models, Proceedings of the 14th ACM Great Lakes symposium on VLSI Boston 2004, Pages: 248 – 253, ISBN:1-58113-853-9
- [32] Andreas Behling: Entwicklung eines minimierenden Fehlerlistengenerators, Diplomarbeit – BTU Cottbus, 2000
- [33] Frommholz, Dirk: Neugestaltung des VHDL-Lademoduls und Konzeption eines Simulatorkerns für FIT zur Unterstützung bidirektionaler Busstrukturen, Studienarbeit – BTU Cottbus, 2006
- [34] Oana Boncalo , Mihai Udrescu , Lucian Prodan , Mircea Vladutiu , Alexandru Amarica: Using Simulated Fault Injection for Fault Tolerance Assessment of Quantum Circuits, 40th Annual Simulation Symposium (ANSS'07), March 2007, pp. 213-220
- [35] Thorsten Grötter, Stan Liao, Grant Martin, Stuart Swan: System Design with SystemC, Kluwer Academic Publishers New York 2002, ISBN 1-4020-60-72-1
- [36] David C. Black, Jack Donovan: SystemC: From the Ground Up, Springer Verlag Berlin 2004, ISBN 1-4020-7988-5
- [37] Jayram Bhasker: A SystemC Primer, Star Galaxy Publishing Allentown 2002, ISBN 0-9650391-8-8
- [38] Wolfgang Müller, Wolfgang Rosenstiel, Jürgen Ruf: SystemC: Methodologies and Applications, Kluwer Academic Publishers Boston 2003, ISBN 1-4020-7479-4
- [39] SystemC Language Working Group: SystemC-Userguide 2.0.1, [www.systemc.org](http://www.systemc.org), 2002
- [40] SystemC Language Working Group: Functional Specification for SystemC 2.0, [www.systemc.org](http://www.systemc.org), 2002
- [41] SystemC Verification Working Group: SystemC Verification Library, <https://www.systemc.org/projects/scv/>, 13.06.2007
- [42] IEEE Standard SystemC® Language Reference Manual: <http://standards.ieee.org/getieee/1666/index.html>, 10.10.2006.
- [43] Synopsys: Describing Synthesizable RTL in SystemCTM, Vers. 1.1. Synopsys Inc., 2002. Available at <http://www.synopsys.com>

- [44] Rolf Drechsler, Görschwin Fey, Christian Genz, Daniel Große: SyCE: An Integrated Environment for System Design in SystemC, 16th IEEE International Workshop on Rapid System Prototyping (RSP), Montreal, 2005
- [45] Daniel Große, Rolf Drechsler: Ein Ansatz zur formalen Verifikation von Schaltungsbeschreibungen in SystemC, it - information technology Number 4, pp. 219-226, August 2003
- [46] Daniel Große, Rolf Drechsler: CheckSyC: An Efficient Property Checker for RTL SystemC Designs, IEEE International Symposium on Circuits and Systems (ISCAS'05), pp. 4167-4170, Kobe 2005
- [47] Modi, S., Wilson, P. R., Brown, A. D. and Chad, J. E.: Behavioral Simulation of Biological Neuron Systems in SystemC (Speech). In Proceedings of IEEE Workshop on Behavioural Modeling and Simulation (BMAS) 2004, San Jose, USA
- [48] SystemC-AMS study group: SystemC-AMS, [www.systemc-ams.org](http://www.systemc-ams.org), 26.02.2007
- [49] Hannes Muhr: Einsatz von SystemC im Hardware/Software-Codesign, Diplomarbeit TU Wien, 2000
- [50] Open Source Initiative (OSI): <http://www.opensource.org>, 28.02.2007
- [51] F. Ghenassia, Ed., Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems. Springer, June 2005, ISBN 0-387-26232-6
- [52] Lukai Cai, Daniel Gajski: Transaction level modeling: an overview, Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS 2003), pp. 19-24
- [53] Daniel Gracia Perez, Gilles Mouchard, Olivier Temam: A New Optimized Implementation of the SystemC Engine Using Acyclic Scheduling, Design, Automation and Test in Europe Conference and Exhibition Volume I 2004, pp. 10552
- [54] R. Le Moigne, O. Pasquier, J-P. Calvez: A Generic RTOS Model for Real-time Systems Simulation with SystemC, Forum DATE'04, 2004, pp. 30082
- [55] Haobo Yu, Andreas Gerstlauer, Daniel Gajski: RTOS scheduling in transaction level models, Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS 2003), pp. 31-36

- [56] Synopsys: Languages - The power of SystemVerilog + SystemC, <http://www.synopsys.com/products/solutions/discovery/languages/languages.html>, 1.3.2007
- [57] Donatella Sciuto, Grant Martin, Wolfgang Rosenstiel, Stuart Swan, Frank Ghenassia, Peter Flake, Johnny Srouji: SystemC and SystemVerilog: Where do They Fit? Where are They Going?, Design, Automation and Test in Europe Conference and Exhibition Volume I, 2004, pp. 10122
- [58] GreenSocs: SystemC modeling: Levels of Abstraction, <http://www.greensocs.com/SystemC/AbstractionLevels>, 1.3.2007
- [59] SystemC Community: SystemC Trends Report 2007, [https://www.systemc.org/projects/sitedocs/document/OSCI\\_Survey\\_Trends\\_Report\\_2007/en/1](https://www.systemc.org/projects/sitedocs/document/OSCI_Survey_Trends_Report_2007/en/1), April 2007
- [60] SystemC Community: SystemC and ESL in 2007, [https://www.systemc.org/projects/sitedocs/document/SystemC\\_Trends\\_Reveal\\_WW\\_Momentum/en/1](https://www.systemc.org/projects/sitedocs/document/SystemC_Trends_Reveal_WW_Momentum/en/1), April 2007
- [61] Doulos Ltd.: SystemC Tutorial, <http://www.doulos.com/knowhow/systemc/tutorial/>, 28.05.2007
- [62] Robert H. Klenke, Ronald D. Williams, James H. Aylor: Parallel-processing techniques for automatic test pattern generation, IEEE Computer, vol. 25, no. 1, pp. 71-84, Jan., 1992.
- [63] Seshu, S., "On an Improved Diagnosis Program", IEEE Trans. on Electronic Comp., Feb. 1965, pp.76-79
- [64] N. Ishiura, M. Ito, S. Yajima: High Speed Fault Simulation Using a Vector Processor, Proc. ICCAD 1987, pp. 10-13, 1987
- [65] Michael J. Quinn: Parallel Programming in C with MPI and OpenMP, McGraw Hill Boston 2003, ISBN 007-123265-6
- [66] Walter Huber: Paralleles Rechnen, R. Oldenbourg Verlag München 1997, ISBN3-486-24383-7
- [67] OpenMP Architecture Review Board: OpenMP, [www.openmp.org](http://www.openmp.org), 10.01.2007.
- [68] John L. Gustafson: Reevaluating Amdahl's law, Communications of the ACM Volume 31, Issue 5, 1988, Pages: 532 – 533, ISSN: 0001-0782
- [69] Intel Corporation: Intel® Compilers, <http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm>, 01.05.2007
- [70] Thomas Rauber, Gudula Rünger: Parallele und verteilte Programmierung, Springer Verlag Berlin 2000, ISBN 3-540-66009-7

- [71] Christian Danegger, Patricia Geugelin-Dannegger: Parallele Prozesse unter UNIX – Simulation und Anwendung bekannter Synchronisationsmethoden in C unter UNIX, Carl Hanser Verlag München 1991, ISBN 3-446-15911-8
- [72] Susan Ragsdale: Parallele Programmierung. Grundlagen, Anwendungen, Methoden, McGraw-Hill London 1992, ISBN 3-89028-397-7
- [73] M.J. Flynn: Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers, 21(9), 1972, 948-960
- [74] Yuan Shi, Reevaluating Amdahl's Law and Gustafson's Law, <http://joda.cis.temple.edu/~shi/docs/amdahl/amdahl.html>, 01.05.2007
- [75] Gene Myron Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, Proceedings of the AFIPS 1967 Spring Joint Computer Conference, S. 483-485, April 1967
- [76] Gerd Meister: Untersuchung des Parallelisierungspotentials diskreter ereignisgesteuerter Simulationen am Beispiel der Schaltkreissimulation, Dissertation, Darmstadt 1999
- [77] Jacek Blazewicz, Klaus Ecker, Brigitte Plateau, Denis Trystram (Ed.): Handbook on Parallel and Distributed Processing, Springer-Verlag Berlin 2000, ISBN 3-54066-441-6
- [78] Message Passing Interface Forum: MPI, <http://www.mpi-forum.org/>, 01.05.2007
- [79] Intel Corporation: Intel® Cluster Toolkit 3.0 for Linux, <http://www.intel.com/cd/software/products/asmo-na/eng/307696.htm>, 15.05.2007
- [80] Argonne National Laboratory: MPICH2: <http://www-unix.mcs.anl.gov/mpi/mpich2/>, 15.05.2007
- [81] LAM team is comprised of students and faculty at Indiana University: LAM/MPI Parallel Computing, <http://www.lam-mpi.org>, 15.05.2007
- [82] Dietrich Rabenstein: Fortran 90, Hanser-Verlag 2001, 3-446-18235-7
- [83] Oak Ridge National Laboratory: PVM, <http://www.csm.ornl.gov/pvm/>, 15.05.2007
- [84] Linus Torvalds, David Diamond: Just for Fun: The Story of an Accidental Revolutionary, HarperCollins Publishers 2002, ISBN 0-06662-073-2
- [85] openSUSE-Communityprojekt: openSUSE Linux, <http://de.opensuse.org/>, 15.05.2007
- [86] Timm Cordes: Parallelisierung des Fehlersimulators Heartless mittels PVM, Diplomarbeit, BTU Cottbus, 2001



- [87] Helmut Herold: Linux-Unix-Systemprogrammierung, Addison-Wesley München, 1999, ISBN 3-8273-1512-3
- [88] F. Brglez, D. Bryan, K. Kozminski, Combinational profiles of Sequential Benchmark Circuits, IEEE ISCAS, 1989, pp. 1929 – 1934
- [89] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits," *Proc. IEEE Int'l Symp. Circuits and Systems*, IEEE Press, Piscataway, N.J., 1985, pp. 695-698
- [90] Xinping Zhu, Wei Qin: Prototyping a fault-tolerant multiprocessor SoC with run-time fault recovery, Proceedings of the 43rd annual conference on Design automation (DAC'06), 2006, Pages: 53 - 56, ISBN 1-59593-381-6
- [91] Neil H. E. Weste, Kamran Eshraghian: Principles of CMOS VLSI Design - A Systems Perspective 2nd edition, Addison Wesley 1994, ISBN 0-201-53376-6
- [92] Parag K. Lala: Practical Digital Logic Design and Test, Prentice Hall Englewood Cliffs 1996, ISBN 0-023-67171-8
- [93] Kothe, R.; Vierhaus, H. T. et al: "Embedded Self Repair by Transistor and Gate Level Reconfiguration", *Proc. IEEE DDECS 2006*, Prag, April 2006, IEEE CS Press, pp. 210-215
- [94] Galke, C.; Kothe, R.; Vierhaus, H. T.: "Logic Self Repair by Transistors and Gate Level Reconfiguration", *Proc. 19th Intern. Conference on Architecture of Computing Systems (ARCS 2006)*, Frankfurt, March 2006, Gesellschaft für Informatik
- [95] V. Krishnaswamy, J. Casas, T. Tetzlaff: A switch level fault simulation environment, proceedings 37th conference on Design automation, June 2000, pp. 780-785
- [96] IEEE Organization: IEEE-1364-Standard, [http://standards.ieee.org/reading/ieee/std\\_public/description/dasc/1364-1995\\_desc.html](http://standards.ieee.org/reading/ieee/std_public/description/dasc/1364-1995_desc.html), 05.04.2007
- [97] John P. Uyemura: Introduction to VLSI Circuits and Systems, Wiley New York, 2002, ISBN 0-471-12704-3
- [98] Christopher A. Ryan, Joseph G. Tront: VHDL switch level fault simulation, European Design Automation Conference 1994, Proceedings Pages: 650-655, ISBN 0-89791-685-9
- [99] Smith, S.P.; Acosta, R.D.: A value system for switch-level modeling, *Design & Test of Computers*, IEEE Volume 7, Issue 3, June 1990 Pages: 33-41
- [100] R. Rajsuman, Y. K. Malaiya, A. P. Jayasumana: On accuracy of switch-level modeling of bridging faults in complex gates, Proceedings of the 24th ACM/IEEE conference on Design automation DAC '87, October 1987

- [101] Volker Hans-Walther Meyer: Test produktionsbedingter Laufzeitfehler in hochintegrierten, digitalen Schaltungen, Dissertation Universität Bremen 2003
- [102] Radhakrishnan, D.; Whitaker, S.R.; Maki, G.K.: Formal design procedures for pass transistor switching circuits, Solid-State Circuits, IEEE Journal of Volume 20, Issue 2, Apr 1985 Page(s): 531 - 536
- [103] Suzuki, M.; Ohkubo, N.; Shinbo, T.; Yamanaka, T.; Shimizu, A.; Sasaki, K.; Nakagome, Y.: A 1.5-ns 32-b CMOS ALU in double pass-transistor logic Solid-State Circuits, IEEE Journal of Volume 28, Issue 11, Nov 1993 Page(s):1145 - 1151
- [104] R. Jacob Baker, Harry W. Li, David E. Boyce: CMOS Circuit Design, Layout, and Simulation, Wiley-IEEE Press 1997, ISBN: 0-780-33416-7
- [105] George Gristede, Wei Hwang: A comparison of dual-rail pass transistor logic families in 1.5V, 0.18 $\mu$ m CMOS technology for low power applications. ACM Great Lakes Symposium on VLSI 2000: 101-106
- [106] Barenco, Adriano, Charles Bennett, et al: Elementary gates for quantum computation, Physical Review A, Vol. 50-5, 1995, pp. 3457-3467
- [107] E. Knill, R. Laflamme, and G.J Milburn, “ A Scheme for Efficient Quantum Computation With Linear Optics”, Nature, pp 46-52, Jan 2001
- [108] G. Schrom, “Ultra Low Power CMOS Technology”, PhD Thesis, Technischen Universität Wien, June 1998
- [109] E. Fredkin, T Toffoli, “Conservative Logic”, International Journal of Theor. Physics, 21(1982),pp.219-253
- [110] J.W. Bruce, M.A. Thornton , L. Shivakumaraiah, P.S. Kokate, X. Li: Efficient Adder Circuits Based on a Conservative Reversible Logic Gate, IEEE Computer Society Annual Symposium on VLSI (ISVLSI'02), April 2002, pp. 0083
- [111] Himanshu Thapliyal, Hamid R. Arabnia: Reversible Programmable Logic Array (RPLA) using Fredkin & Feynman Gates for Industrial Electronics and Applications, Published in Proceedings of the International Conference on Embedded Systems and Applications(ESA'06), Las Vegas, June 2006
- [112] Mei-Chen Hsueh, Timothy K. Tsai, Ravishankar K. Iyer: Fault Injection Techniques and Tools, IEEE Computer April 1997 (Vol. 30, No. 4). pp. 75-82
- [113] Clark, J. A., Pradhan, D. K.: Fault Injection: A Method for Validating Computer System Dependability, IEEE Computer Vol. 28, No. 6, June 1995, pp. 47-56

- [114] F. Bruschi, M. Chiamenti, F. Ferrandi, D. Sciuto: Error Simulation Based on the SystemC Design Description Language, Design, Automation and Test in Europe Conference and Exhibition (DATE'02) 2002, pp. 1135
- [115] Francesco Bruschi , Fabrizio Ferrandi, Donatella Sciuto: A Framework for the Functional Verification of SystemC Models, International Journal of Parallel Programming, Springer Netherlands 2005, Volume 33, Number 6 / December, 2005, pp. 667-695
- [116] F. Ferrandi, M. Rendine, D. Sciuto: Functional Verification for SystemC Descriptions Using Constraint Solving, Design, Automation and Test in Europe Conference and Exhibition (DATE'02) 2002, pp. 0744
- [117] Ferrandi, F., Ferrara, G., Scuito, D., Fin, A., Fummi, F.: Functional Test Generation for Behaviorally Sequential Models. Proc. Design, Automation and Test in Europe, 2001, pp. 403-410.
- [118] K. Rothbart, U. Neffe, Ch. Steger, R. Weiss, E. Rieger, A. Muehlberger: High Level Fault Injection for Attack Simulation in Smart Cards, Found in: 13th Asian Test Symposium (ATS'04), November 2004, pp. 118-121
- [119] Alessandro Fin, Franco Fummi, Graziano Pravadelli: AMLETO: A Multi-language Environment for Functional Test Generation, Proceedings International Test Conference 2001, pp. 821
- [120] P.A. Wilsey, D.E. Martin, K. Subramani: SAVANT/TyVIS/WARPED: components for the analysis and simulation of VHDL, Verilog HDL Conference and VHDL International Users Forum Santa Clara, CA, USA 1998. IVC/VIUF. Proceedings, pp. 195-201, ISBN 0-8186-8415-1
- [121] P.A. Wilsey, D.E. Martin, H. Hirsch: The SAVANT project, Aerospace and Electronics Conference, 1998. NAECON 1998. Proceedings pp. 537-544, ISBN 0-7803-4449-9
- [122] Wilsey et al. : SAVANT: VHDL Analysis Tools, <http://www.eecs.uc.edu/~paw/savant/>, 19.02.2007
- [123] Alfredo Benso (Herausgeber), P. Prinetto (Herausgeber): Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation (Frontiers in Electronic Testing), Springer US, 2003, ISBN 1-402-07589-8
- [124] J. Gracia, J.C. Baraza, D. Gil, P.J. Gil: Comparison and Application of Different VHDL-Based Fault Injection Techniques, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'01), October 2001, pp. 0233.

- [125] J.C. Baraza, J. Gracia, D. Gil, P.J. Gil: Improvement of fault injection techniques based on VHDL code modification, High-Level Design Validation and Test Workshop, 2005. Tenth IEEE International, November 2005, pp. 19-26
- [126] M. Rimén, J. Ohlsson, J. Karlsson, et al.: Design Guidelines of a VHDL-based Simulation Tool for the Validation of Fault Tolerance 1997
- [127] Jean-Claude Laprie (ed.): Dependability: Basic Concepts and Terminology, Springer-Verlag, Wien 1992, ISBN 3-211-82296-8
- [128] Johnson, W. B.: Design and Analysis of Fault Tolerance Digital Systems” Addison-Wesley 1989
- [129] Todd A. Delong, Barry W. Johnson, Joseph A. Profeta Iii: A Fault Injection Technique for VHDL Behavioral-Level Models, IEEE Design and Test of Computers, December 1996, pp. 24-33
- [130] Jean Arlat et al. Fault injection for dependability validation: A methodology and some applications. IEEE Transactions on Software Engineering, 16(2):166–182, 1990
- [131] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G.H. Leber, Comparison of physical and software-implemented fault injection techniques. Computers, IEEE Transactions on, 2003. 52(9): pp. 1115-1133.
- [132] E. Jenn, J. Arlat, M. Rimen. J. Ohlsson, J. Karlsson: Fault Injection into VHDL Models: The MEFISTO Tool, Proc. 24th International Symposium Fault-Tolerant Computing, IEEE, 1994 pp. 66-75.
- [133] Stefan Freinatis: Towards Comparability in Evaluating the Fault-Tolerance of Safety-Critical Embedded Software, Dissertation, Universität Duisburg–Essen 2005
- [134] A. Benso, P. L. Civera, M. Rebaudengo, M. Sonza Reorda, A. Ferro: A Hybrid Fault Injection Methodology for Real Time Systems, Digest of FastAbstracts: FTCS-28, The 28th Annual International Symposium on Fault-Tolerant Computing, June 1998, Munich (Germany), pp. 74-75
- [135] J. Vinter, J. Aidemark, D. Skarin, R. Barbosa, P. Folkesson, J. Karlsson: An Overview of GOOFI - A Generic Object-Oriented Fault Injection Framework, Technical Report No. 05-07, University of Göteborg 2005
- [136] H. Madeira, M. Rela, J. G. Silva: RIFLE: A General Purpose Pin-Level Fault Injector", Proc. First European Dependable Computing Conference (EDCC-1), Berlin, Germany, Springer Verlag, 4.-6. Oktober, 1994, S. 199-216.

- [137] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo, Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms. *IEEE Micro*, 1994. 14(1): pp. 8-23
- [138] J. Karlsson, U. Gunneflo, P. Liden, and J. Torin. "Two fault injection techniques for test of fault handling mechanisms", in *Proceedings. International Test Conference 1991 (IEEE Cat. No.91CH3032-0)*. Nashville, TN, USA. 26-30 October, 1991. pp. 140-149
- [139] J.R. Samson, Jr., W. Moreno, and F. Falquez. "A technique for automated validation of fault tolerant designs using laser fault injection (LFI)", in *Proceedings of 28th International Symposium on Fault Tolerant Computing*. Munich, Germany. 23-25 June, 1998. pp. 162-167.
- [140] S.A. Aftabjahi, Z. Navabi: Functional Fault Simulation of VHDL Gate Level Models, *VHDL International User's Forum (VIUF '97)*, 1997, pp. 18
- [141] Berkeley-SPICE: The Spice Page, <http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/>, 01.03.2007
- [142] The MathWorks, Inc.: Matlab, <http://www.mathworks.com/products/matlab/>, 01.03.2007
- [143] Babak Rahbaran, Andreas Steininger, Thomas Handl: Built-in Fault Injection in Hardware - The FIDYCO Example, 2nd IEEE International Workshop on Electronic Design, Test and Applications, January 2004, pp. 327
- [144] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero: RT-level Fault Simulation Techniques based on Simulation Command Scripts, *DCIS2000: XV Conference on Design of Circuits and Integrated Systems*, Le Corum, Montpellier, November 21-24, 2000, pp. 825-830
- [145] Frank Balbach: Analyse von Hardwarefehlern in Vermittlungseinheiten digitaler Netze, Dissertation, Universität Erlangen 2000
- [146] Volkmar Sieh, Oliver Tschache, Frank Balbach: VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions, 27th International Symposium on Fault-Tolerant Computing (FTCS '97), June 1997, pp. 32
- [147] R. De Millo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", *IEEE Computer*, vol. 11, No. 4, pp. 34-41, 1978
- [148] Ghassan Al Hayek, Chantal Robach: From Specification Validation to Hardware Testing: A Unified Method, *Proceedings International Test Conference 1996*, pp. 885

- [149] Mathieu Scholivé, Vincent Beroulle, Chantal Robach, Marie-Lise Flottes, Bruno Rouzeyre: Mutation Sampling Technique for the Generation of Structural Test Data. DATE 2005, pp. 1022-1023
- [150] Alessandro Fin, Franco Fummi: A VHDL error simulator for functional test generation, Proceedings of the conference on Design, automation and test in Europe, March 2000, pp. 390-395
- [151] Matthias Pflanz: On-Line Error Detection and Fast Recover Techniques for Dependable Embedded Processors, Springer Verlag Berlin 2002, ISBN 3-540-43318-X
- [152] P. C. Ward, J. R. Armstrong: Behavioral fault simulation in VHDL, Proceedings of the 27th ACM/IEEE conference on Design automation Orlando, Florida, United States, Pages: 587 - 593, 1990, ISBN 0-89791-363-9
- [153] J.P. Roth. Diagnosis of Automata Failure: A Calculus and a Method. IBM Journal of Research and Development, 10:278–291, October 1966.
- [154] A. Avizienis: Fault-tolerant systems, IEEE Transactions on Computer, December 1976, pp. 1304-1311
- [155] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, Carl Landwehr: Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing, vol. 01, no. 1, pp. 11-33, JANUARY-MARCH 2004
- [156] F. Balbach (ed.), M. Dal Cin, A. Hein, J. Meixner, B. Rümenapp, V. Sieh, J. Stiborsky, O. Tschäche: Simulationsbasierte Zuverlässigkeitsanalyse, Internal Report No.: 6/96, Universität Erlangen 1996
- [157] G. Kemnitz: Test und Verlässlichkeit von Rechnern, TU Clausthal, Institut für Informatik, 2000
- [158] M. Abramovici, M.A. Breuer, A.D. Friedman: Digital System Testing and Testable Design, IEEE Press, New York, ISBN 0-7803-1062-4, 1990
- [159] Armstrong J.R., Lam F.-S., Ward P.C.: Test generation and Fault Simulation for Behavioral Models”, In Performance and Fault Modeling with VHDL (J. M. Schoen ed.) S.240-303, Englewood Cliffs, Prentice-Hall, 1992
- [160] Robert Baumann: Soft Errors in Advanced Computer Systems, IEEE Design & Test of Computers 2005, pp. 256-266
- [161] R. D. Eldred, “Test Routines Based on Symbolic Logical Statements,” Journal of the ACM, vol. 6, no. 1, pp. 33–36, Jan. 1959.
- [162] Ketan N. Patel , John P. Hayes , Igor L. Markov: Fault Testing for Reversible Circuits, 21st IEEE VLSI Test Symposium, April 2003, pp. 410

- [163] K. C. Y. Mei: Bridging and stuck-at faults. IEEE Transaction on Computers, C-32(7): 720–727, 1974
- [164] J.M. Acken: Testing for Bridging Faults (Shorts) in CMOS Circuits, 1983 Design Automation Conference, pp. 717-718, 1983
- [165] K.J. Lee, M.A. Breuer: On detecting Single and Multiple Bridging Faults in CMOS Circuits Using Current Supply Monitoring Method, 1990 International Conference on Circuits and Systems, May 1990
- [166] H.T. Vierhaus, W. Meyer, U. Gläser: CMOS Bridges and Resistive Transistor Faults: IDDQ versus Delay Effects, Int. Test Conference 1993, Baltimore, USA, pp. 83-91
- [167] D. B. Lavo, T. Larrabee, and B. Chess, "Beyond the Byzantine Generals: Unexpected Behavior and Bridging Faults Diagnosis," in Proc. of the IEEE Intl. Test Conf., pp. 611-619, Oct. 1996
- [168] Don Shaw, Dhamin Al-Khalili, Come Rozon: Accurate CMOS Bridge Fault Modeling With Neural Network-Based VHDL Saboteurs, 2001 International Conference on Computer-Aided Design (ICCAD '01), 2001, pp. 531
- [169] J. M. Acken, S. D. Millman: Accurate modeling an simulation of bridging faults, Proceedings of the custom Integrated Circuits Conference, 1991
- [170] J. M. Acken, S. D. Millman: Fault model evolution for diagnosis: accuracy vs. precision, Proceedings of the custom Integrated Circuits Conference, 1992
- [171] Erik Markert, Hailu Wang, Göran Herrmann, Ulrich Heinkel: Kostenmodellierung mit SystemC/SystemC-AMS, Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS 2007), Mai 2007
- [172] Sar-Dessai, V.; Walker, D.M.H.: Accurate fault modeling and fault simulation of resistive bridges, Defect and Fault Tolerance in VLSI Systems, 1998. Proceedings., 1998 IEEE International Symposium on Volume , Issue , 2-4 Nov 1998 Page(s):102 - 107
- [173] R.Kothe, H. T. Vierhaus: Flip-Flops and Scan-Path Elements for Nanoelectronics, IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, Krakow, April 2007, pp. 307
- [174] Cristian Constantinescu: Impact of Deep Submicron Technology on Dependability of VLSI Circuits, Proceedings of the International Conference on Dependable Systems and Networks (DSN '02), 2002
- [175] Dirk Weiler: Selbsttest und Fehlertoleranz mit zugelassener milder Degradation in integrierten CMOS Sensorsystemen, Gerhard-Mercator-Universität - Gesamthochschule Duisburg, Dissertation, 2001

- [176] Stroustrup, Bjarne: Die C++-Programmiersprache, Verlag Addison-Wesley Bonn, ISBN 3-89319-386-3
- [177] Dirk Louis: C / C++ Professionelles Kompendium, Markt und Technik München 2004, ISBN 3-827-26812-5
- [178] SystemVerilog, <http://www.systemverilog.org>, 1.3.2007
- [179] Forte Design Systems: Cynthesizer, <http://www.forteds.com/products/cynthesizer.asp>, 31.10.2006
- [180] Synopsys, Inc.: Describing Synthesizable RTL in SystemC Version 1.1, 2002
- [181] SystemC Synthesis Working Group: SystemC\_Synthesizable\_Subset\_Draft.1.1.18, <https://www.systemc.org/download/5/47/73/128/>, 21.05.2007
- [182] Celoxica: Agility Compiler, <http://www.celoxica.com/products/agility/default.asp>, 2.3.2007
- [183] Petra Nootz, Franz Morick: C/C++ Referenz, Franzis-Verlag Poing 2000, ISBN 3-7723-6355-5
- [184] Perl: [www.perl.org](http://www.perl.org), 02.04.1007
- [185] V. Toth, D. Louis: Visual C++ 6 Kompendium. Umfassende Referenz für Programmierer, Markt+Technik Verlag, 1998, ISBN-10: 3-8272-5467-1
- [186] CodeGear: Delphi, <http://www.codegear.com/Products/tabid/98/Default.aspx>, 02.04.2007
- [187] Boost Community: boost c++ Libraries, <http://www.boost.org/>, 02.05.2007
- [188] Davis Chapman: Visual C++ 6, Markt und Technik, 2004, ISBN 3-8272-6811-7
- [189] Gunther Lehmann, Bernhard Wunder, Manfred Selz: Schaltungsdesign mit VHDL. Synthese, Simulation und Dokumentation digitaler Schaltungen, Franzis Verlag GmbH 1994, ISBN 3-772-36163-3
- [190] Teich, Jürgen: Digitale Hardware-, Software-Systeme, Synthese und Optimierung, Springer Berlin 1997, ISBN 3-540-62433-3
- [191] R. Walker, D. Thomas: A Model of Design Representation and Synthesis, proceedings of the 22nd Design Automation Conference, Las Vegas, 1985, pp. 453-459
- [192] IEEE Organization: [www.ieee.org](http://www.ieee.org), 01.03.2007
- [193] Bailey, B., Gajski, D.: RTL semantics and methodology, ISSS '01, Montréal 2001, pp. 69-74
- [194] edacentrum e. V. und edacentrum GmbH: Electronic Design Automation Centrum, [www.edacentrum.de](http://www.edacentrum.de), 20.06.2007



- [195] Peter Wintermayr: Schwerpunkt EDA Tools, Markt & Technik 10/2007, WEKA Fachzeitschriften Verlag GmbH, 2007, ISSN 0344-8843
- [196] The Object Management Group™: UML® Resource Page, <http://www.uml.org/>, 21.06.2007
- [197] Jan Hendrik Hausmann: Dynamische Metamodellierung zur Spezifikation einer operationalen Semantik von UML, Diplomarbeit - Universität Paderborn, 2001
- [198] F. Martinolle, C. Dawson, D. Corlette, M. Floyd: Interoperability of Verilog/VHDL Procedural Language Interfaces to Build a Mixed Language GUI, DATE '99, München, 1999
- [199] Francoise Martinolle, Uma Parvathy: Mixed Language Design Data Access: Procedural Interface Design Considerations, VHDL International Users Forum Fall Workshop (VIUF'00), October 2000, pp. 95
- [200] Martin Radetzki: Mixed-language simulation for SystemC and VHDL, 11. ESCUG, München 2005
- [201] Terence Chan: Multithreaded, mixed hardware description languages logic simulation on engineering workstations, U.S. Patent No. 6,466,898, Oct 2002
- [202] B. Niemann, M. Speitel: RTL Design-Flow with SystemC in an Industrial Design Project – Modeling a GPS Receiver Using SystemC, SNUG Europe 2001
- [203] P. Sauge, G. Thuau: Integrating of Verilog-HDL and VHDL Languages in the SMASH (tm) Mixed-signal Multi-level Simulator, International Verilog HDL Conference and VHDL International Users Forum, March 1998, pp. 2
- [204] J. Lawrence, C. Ussery INCA: a next-generation architecture for simulation, IEEE International Verilog HDL Conference (IVC '96), March 1996, pp. 12
- [205] Pedram A. Riahi, Zainalabedin Navabi, Fabrizio Lombardi: Simulating Faults of Combinational IP Core-based SOCs in a PLI Environment, 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05), October 2005, pp. 389-397
- [206] Anslab: Interfacing SystemC to HDL using PLI/VPI & FLI, [https://www.systemc.org/docman2/ViewCategory.php?group\\_id=4&category\\_id=15](https://www.systemc.org/docman2/ViewCategory.php?group_id=4&category_id=15), 2002
- [207] Bill Dittenhofer and Dr. Greg Tumbush: Design and Verification of a Processor Using VHDL, Verilog, System C and C++, Design and Verification Conference, San Jose 2004
- [208] Shields, J.: Modeling foreign architectures with VHPI, VHDL International Users Forum Fall Workshop, Proceedings Volume , 2000 Page(s): 100 - 107

- [209] Accelera: Accellera approves new VHDL standard, <http://www.accellera.org/pressroom/2006/AcelleraVHDL100906.pdf>, 21.04.2007
- [210] C. Rousselle, M. Pflanz, A. Behling, T. Mohaupt, H. Vierhaus: A Register-Transfer-Level Fault Simulator for Permanent and Transient Faults in Embedded Processors, Design, Automation, and Test in Europe (DATE '01), March 2001, pp. 0811
- [211] Altera Corp.: Quartus II Development Software Version 7.0 Literature, <http://www.altera.com/literature/lit-qts.jsp>, 25.4.2007
- [212] Microsoft Corporation: MS-DOS 5.0 interne und externe Befehle, <http://support.microsoft.com/kb/71986>, 25.04.2007
- [213] Rüdiger Brause: Betriebssysteme. Grundlagen und Konzepte, Springer Verlag Berlin 1998, ISBN 3-540-62929-7
- [214] William Stalling: Operating Systems – Fourth International Edition, Prentice Hall 2001, ISBN 0-13-032986-6
- [215] POSIX-Threads: <http://sources.redhat.com/pthreads-win32/>, 09.03.2006.
- [216] Lawrence Livermore National Laboratory: POSIX Threads Programming, <http://www.llnl.gov/computing/tutorials/pthreads/>, 28.04.2007
- [217] Charles Petzold: Windows-Programmierung 5. Auflage, Microsoft Press Deutschland 2000, ISBN 3-86063-487-9
- [218] Jeffrey Richter: Microsoft Windows Programmierung für Experten 4. Auflage, Microsoft Press Deutschland 2000, ISBN3-86063-615-4
- [219] Albert Lauchner: Hyper-Threading: Optimierungen und Fallen, <http://www.tecchannel.de/entwicklung/programmierung/402048/>, 29.01.2003
- [220] Intel: Processors - Hyper-Threading Technology, <http://www.intel.com/support/processors/sb/cs-017343.htm>, 30.04.2007
- [221] Sandip Kundu: Pitfalls of Hierarchical Fault Simulation, IEEE Transactions on Computer-Aided Design of ICs and Systems, Vol. 23, No. 2 Feb. 2004
- [222] Misera, S.; Vierhaus, H. T.: "FIT - A Parallel Hierarchical Fault Simulator", Intern. Conference on Parallel Computing in Electrical Engineering (PARELEC 2004), Dresden, Ed. R. Merker, IEEE Comp. Society Press, pp. 289-296, ISBN 0-7695-2080-4
- [223] Misera, S.; Vierhaus, H. T.; Breitenfeld, L.; Sieber, A.: "A Mixed Language Fault Simulation of VHDL and SystemC", 9th Euromicro Conference on Digital System Design 2006, Cavtat/Croatia, Ed. V. Muthukumar, IEEE Comp. Society Press, pp. 275-279, ISBN 0-7695-2609-8

- [224] Silvio Misera, Heinrich Theodor Vierhaus, Andre Sieber: Fault Injection Techniques and their Accelerated Simulation in SystemC, 10th Euromicro Conference on Digital System Design 2007 , Lübeck, August 2007, ISBN 0-7695-2978-X
- [225] Misera, S.; Sieber, A.; Breitenfeld, L.; Vierhaus, H. T.: "Eine Mixed-Language-Fault-Simulation von VHDL- und SystemC-Modellen", Fraunhofer Institut Integrierte Schaltungen - Dresdner Arbeitstagung Schaltungs- und Systementwurf DASS 2006
- [226] Misera, S.: "Hierarchische Fehlersimulation mit effektiven SystemC-Modellen", 1st Cooperation Workshop of Computer Science 2006, Cottbus, Computer Science Reports BTU Cottbus, ISSN: 1437-7969
- [227] Misera, S.; Sieber, A.: "Hardware-nahe Fehlersimulation mit effektiven SystemC-Modellen", 10. GI/ITG/GMM Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", Erlangen, März 2007, ISBN 3-8322-5956-2
- [228] Misera, S.; Sieber, A.: "Fehlerinjektionstechniken in SystemC-Beschreibungen mit Gate- und Switch-Level-Verhalten", Fraunhofer Institut Integrierte Schaltungen - Dresdner Arbeitstagung Schaltungs- und Systementwurf DASS 2007, ISBN-13: 978-3-940046-28-4
- [229] Galke, C.; Misera, S.; Fröschke, H.; Vierhaus, H. T.: "Eine Simulationsumgebung zur Validierung des Fehlerverhaltens für Prozessor-basierte Systeme " - Poster, Proc. 17. ITG-GI-GMM-Workshop "Test und Zuverlässigkeit von Schaltungen und Systemen", Innsbruck, Febr./März 2005, Ed. S. Hellebrand
- [230] Galke, C.; Misera, S.; Fröschke, H.; Vierhaus, H. T.: "Parallele Hierarchische Fehlersimulation zur Validierung des Fehlerverhaltens für SoCs", GI/ITG/GMM Workshop "Modellierung und Verifikation", April 2005, München

